НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«До захисту допущено»
Науковий керівник кафедри
_____ І.А. Дичка
«___»_____2019 р.

Дипломний проект

на здобуття ступеня бакалавра

з напряму підготовки 6.050103 «Програмна інженерія»

на тему: «Компілятор мови ASAMPL»

Виконав:
студент IV курсу, групи КП-52
Песчанський Владислав Юрійович _____

Керівник:
Доцент, к.т.н.,
Сулема Є. С. _____

Консультант з нормоконтролю:
Доцент кафедри ПЗКС, к.т.н.,
Онай М.В. _____

Рецензент:
В.о. завідувача кафедри ММСФ ІПСА, к.т.н., доцент.
Тимощук О.Л. _____

Засвідчую, що у цьому дипломному проекті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2019 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет прикладної математики
Кафедра програмного забезпечення комп'ютерних систем

Рівень вищої освіти – перший (бакалаврський)

Напрям підготовки – 6.050103 «Програмна інженерія»

ЗАТВЕРДЖУЮ
Науковий керівник кафедри
_____ І.А. Дичка
«___»_____2018 р.

ЗАВДАННЯ
на дипломний проект студенту

Песчанському Владиславу Юрійовичу

1. Тема проекту «Компілятор мови ASAMPL», керівник проекту Сулема Євгенія Станіславівна, к.т.н., доцент, затверджені наказом по університету від «___»_____ 2019 р. №_____

2. Термін подання студентом проекту «19» червня 2019 р.

3. Вихідні дані до проекту: див. Технічне завдання.

4. Зміст пояснювальної записки:
   1) аналіз існуючих рішень;
   2) обґрунтування вибору засобів реалізації;
   3) розроблення алгоритмів та програмних модулів;
   4) аналіз розробленого компілятора мови ASAMPL.

5. Перелік обов'язкового графічного матеріалу:
   1) Діаграма діяльності компілятора мови ASAMPL (креслення);
   2) Діаграма діяльності модулю лексичного аналізу (креслення);
   3) Узагальнена схема розташування елементів AST дерева (плакат);
   4) Діаграми класів розробленого компілятора (плакат).

6. Консультанти розділів проекту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Онай М.В., доцент		

7. Дата видачі завдання «31» жовтня 2018 р.

Календарний план

№ з/п	Назва етапів виконання дипломного проекту	Термін виконання етапів проекту	Примітка
1.	Вивчення літератури за тематикою проекту	14.11.2018	
2.	Розроблення та узгодження технічного завдання	28.11.2018	
3.	Розроблення архітектури компілятора мови ASAMPL	15.12.2018	
4.	Підготовка матеріалів першого розділу дипломного проекту	30.12.2018	
5.	Підготовка матеріалів другого розділу дипломного проекту	03.02.2019	
6.	Розроблення алгоритмів компілятора мови ASAMPL	20.02.2019	
7.	Програмна реалізація компілятора мови ASAMPL	10.03.2019	
8.	Тестування компілятора мови ASAMPL	20.04.2019	
9.	Підготовка матеріалів третього розділу дипломного проекту	10.05.2019	
10.	Підготовка матеріалів четвертого розділу дипломного проекту	20.05.2019	
11.	Підготовка презентації, креслень та плакатів	01.06.2019	
12.	Оформлення документації дипломного проекту	10.06.2019	

Студент Песчанський В.Ю.

Керівник проекту Сулема Є.С.

АНОТАЦІЯ

Цей дипломний проект присвячений розробленню компілятора мови ASAMPL. В рамках виконання проекту здійснено порівняльний аналіз існуючих інструментів розроблення та обрано засоби, що найкраще підходять для вирішення поставленої задачі. У дипломному проекті розроблено та описано основні алгоритми, які виконуються програмним застосунком для забезпечення компіляції виконуваних файлів програмного коду мовою ASAMPL, а саме: алгоритм лексичного аналізу, алгоритм синтаксичного аналізу вхідного потоку лексем і алгоритм інтерпретації абстрактного синтаксичного дерева у виконуваний машинний код.

Розроблений компілятор спрощує роботу з програмним кодом, написаним мовою ASAMPL, та дозволяє вільно відкривати, оброблювати та зберігати мультимедійні дані, що надходять на вхід комп'ютерної системи від різноманітних джерел: від простих файлів до даних, що надходять з давачів, які працюють у реальному часі.

Отже, даний програмний застосунок дозволяє полегшити розроблення спеціалізованого мультимедійного програмного забезпечення мовою ASAMPL.

ABSTRACT

This diploma project is devoted to the development of the language compiler ASAMPL. As a part of the project, a comparative analysis of existing development tools is done and the tools, which are best suited for solving the task, are selected. In the diploma project, the basic algorithms, which are xecuted by a software application for compiling executable files in the ASAMPL language code, are developed and described, namely: the lexical analysis algorithm, the algorithm for syntactic analysis of the input lexemes stream and the algorithm for interpreting the abstract syntax tree into the executable machine code.

The developed compiler makes it easy to work with a program code written in ASAMPL language, and allows freely to open, process and store multimedia data coming to the computer system from various sources: from simple files to data coming from real-time sensors.

Consequently, this software application facilitates the development of specialized multimedia software in the language ASAMPL.

ДП.045480-01-90 Компілятор мови ASAMPL. Відомість проекту

Позначення	Найменування	Кіл-ть	Примітка
	Документація проекту		
ДП.045480-02-91	Компілятор мови	5	
	ASAMPL. Технічне		
	завдання		
ДП.045480-03-81	Компілятор мови	62	
	ASAMPL. Пояснювальна		
	записка		
ДП.045480-04-51	Компілятор мови	4	
	ASAMPL. Програма та		
	методика тестування		
ДП.045480-05-34	Компілятор мови	5	
	ASAMPL. Керівництво		
	користувача		
ДП.045480-06-99	Компілятор мови	1	
	ASAMPL. Діаграма		
	діяльності модулю		
	лексичного аналізу		
ДП.045480-07-99	Компілятор мови	1	
	ASAMPL. Діаграма		
	діяльності компілятору		
	мови ASAMPL		
ДП.045440-08-98	Компілятор мови	1	
	ASAMPL. Компакт-диск.		

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

"ЗАТВЕРДЖЕНО"

Науковий керівник кафедри

_____ І.А. Дичка

"___" _____ 2018 р.

КОМПІЛЯТОР МОВИ ASAMPL

Технічне завдання

ДП.045480-02-91

"ПОГОДЖЕНО"

Керівник проекту:

_____ Є.С. Сулема

Нормоконтроль:

_____ М.В. Онай

Виконавець:

_____ Песчанький В.Ю.

2018

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ГАЛУЗЬ ЗАСТОСУВАННЯ

Назва розробки: Компілятор мови ASAMPL.

Галузь застосування: інформаційні технології.

2. ПІДСТАВА ДЛЯ РОЗРОБЛЕННЯ

Підставою для розроблення є завдання на дипломне проектування, затверджене кафедрою програмного забезпечення комп'ютерних систем Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського» (КПІ ім. Ігоря Сікорського).

3. ПРИЗНАЧЕННЯ РОЗРОБКИ

Розробка призначена для обробки програм, написаних згідно з формальним описом граматики мови ASAMPL з метою полегшити програмну обробку мультимедійних даних для пересічного користувача.

4. ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ

Компілятор повинен забезпечувати такі основні функції:

1. Виконання програм, написаних згідно з формальним описом граматики мови ASAMPL;
2. Оброблення та відображення помилок, виникаючих у процесі обробки поданих програм.

Додаткові вимоги:

1. Стійкість до виникаючих під час виконання тексту програм помилок;
2. Інформативний звіт про виникаючі помилки.

5. ВИМОГИ ДО ПРОЕКТНОЇ ДОКУМЕНТАЦІЇ

У процесі виконання проекту повинна бути розроблена наступна документація:

1. пояснювальна записка;
2. програма та методика тестування;
3. керівництво користувача;
4. креслення:
    - «Діаграма діяльності компілятору мови ASAMPL»;
    - «Діаграма діяльності модулю лексичного аналізу».

6. ЕТАПИ ПРОЕКТУВАННЯ

Вивчення літератури за тематикою роботи…….14.11.2018

Розроблення та узгодження технічного завдання……..28.11.2018

Розроблення архітектури компілятора мови ASAMPL……15.12.2018

Підготовка першого розділу дипломного проекту……30.12.2018

Підготовка другого розділу дипломного проекту…….03.02.2019

Розроблення алгоритмів компілятора мови ASAMPL……..20.02.2019

Програмна реалізація компілятора мови ASAMPL……...10.03.2019

Тестування компілятора мови ASAMPL…….....20.04.2019

Підготовка третього розділу дипломного проекту……....10.05.2019

Підготовка четвертого розділу дипломного проекту…….20.05.2019

Підготовка презентації, креслень та плакатів……01.06.2019

Оформлення технічної документації проекту……..…...10.06.2019

7. ПОРЯДОК ТЕСТУВАННЯ РОЗРОБКИ

Тестування розробленого програмного продукту виконується відповідно до “Програми та методики тестування”.

Факультет прикладної математики Кафедра програмного забезпечення комп'ютерних систем

"ЗАТВЕРДЖЕНО"
Науковий керівник кафедри
_____ І.А. Дичка
"___" _____ 2019 р.

КОМПІЛЯТОР МОВИ ASAMPL

Пояснювальна записка

ДП.045480-03-81

"ПОГОДЖЕНО"

Керівник проекту:

_____ Є.С. Сулема

Нормоконтроль:

_____ М.В. Онай

Виконавець:

_____ Песчанський В.Ю.

2019

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

Фреймворк – інфраструктура програмних рішень, що полегшує розробку складних систем. Спрощено дану інфраструктуру можна вважати своєрідною комплексною бібліотекою, але при цьому вона має ряд обмежень, що задають правила створення структури проекту та написання коду.

DRM (*Digital restrictions managemen*) – термін, який використовується для посилання на технології авторизації, що застосовуються виробниками апаратного забезпечення, видавцями, власниками авторських прав або приватними особами в першу чергу для обмеження використання цифрової інформації та носіїв.

DirectX – це набір API функцій, розроблених для простого і ефективного вирішення завдань, пов'язаних з ігровим і відеопрограмуванням під Microsoft Windows.

API (*Application programming interface*) – набір визначень підпрограм, протоколів взаємодії та засобів для створення програмного забезпечення.

TIOBE – показник популярності мов програмування, створений і підтримуваний компанією TIOBE.

ARM (*Advanced RISC machine*) – 32-бітна RISC архітектура процесорів, яку розробила компанія ARM Limited.

RISC (*Reduced instruction set computing*) – архітектура процесорів зі скороченим набором команд.

CLI (*Common language infrastructure*) – специфікація загальномовної інфраструктури.

BCL – Стандартна бібліотека класів платформи «.NET Framework».

Лексер – програма чи функція, що перетворює вхідну послідовність символів у послідовність лексем (груп символів, що відповідають певним шаблонам та мають певний тип).

Парсер – компонент програми, що виконує перетворення вхідної послідовності символів в структурований формат згідно з певною формальною граматикою.

Транслятор – програма або технічний засіб, який виконує перетворення чи іншу обробку текстів програм.

Інтерпритатор – програма чи технічні засоби, необхідні для виконання інших програм, вид транслятора, який здійснює пооператорну (покомандну, построкову) обробку, перетворення у машинні коди та виконання програми або запиту.

EBNF (*Extended Backus–Naur form*) – це розширена форма способу запису правил контекстно-вільної граматики.

AST (*Abstract syntax tree*) – це скінченна множина, позначене і орієнтоване дерево, в якому внутрішні вершини співставлені з відповідними операторами мови програмування, а листя з відповідними операндами.

ВСТУП

На сьогоднішній день мультимедіа являє собою одну з найбільш стрімко розвиваючихся комп'ютерних галузей серед усіх. В першу чергу це пов'язано зі стрімким розвитком пристроїв відображення інформації. Усього за чверть століття вони еволюціонували від примітивних маленьких променевих екранів до таких виробів майбутнього як навіть голограми які майже не можна відрізнити від реальності.

Майже усі винаходи у апаратній та програмній частині мультимедіа знайшли та продовжують знаходити широке застосування у повсякденному житті кожної людини. Кінотеатри, реклама, ігрова та розважальні індустрії, все це використовує уже устатковані мультимедійні технології та широко розвивають нові галузі. Зараз розвиток відбувається так швидко, що вже й важко сказати, яка з поки-що фантастичних речей завтра може стати частиною повсякденного життя.

Задля поліпшення продуктивності використання оновленої апаратної частини, слідом швидко розвивається і різноманітна програмна частина, націлена на опанування нових можливостей щойно розроблених різних архітектур, а також для поліпшення продуктивності вже існуючих алгоритмів оброблення даних.

Особливо слід звернути увагу на бурхливий розвиток такої частина програмного забезпечення, як штучні нейронні мережі. Їх використання в першу чергу значно полегшило розпізнавання графічних образів, в тому числі і в реальному часі.

Отже, можна зробити висновок, що з кожним роком алгоритми обробки мультимедійної інформації стають все складнішими і складнішими, а форматів різноманітних мультимедійних даних, які кожен раз намагаються вдосконалити вже існуючі способи їх збереження, стає все більше і більше, що ускладнює їх обробку.

Таким чином для звичайного користувача апаратного або програмного забезпечення з часом задача обробки та збереження мультимедійних даних стає все більш складною і незрозумілою.

Отже, на основі вищезазначеного можна сформулювати головну мету даного дипломного проекту – розроблюваний дипломний проект призначено для розвитку та частково створення ядра мови ASAMPL – предметно орієнтованої мови програмування основним завданням якої є полегшення для пересічного користувача задачу обробки різноманітних мультимедійних даних.

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

1.1. Загальне застосування мультимедійних технологій

Все більш зростаючий обсяг мультимедійних даних, що отримуються у результаті збільшеної у багато разів складності сучасних алгоритмів, розвитку систем машинного навчання та набагато збільшившись можливостей сучасної інтернет мережі, а також значно збільшена кількість форматів цих даних, що буквально з кожним роком поповнюється більш досконалими аналогами та кращими альтернативами починає невідмінно потребувати все досконаліші та досконаліші методи їх збереження, аналізу та подальшої обробки.

В свою чергу це може потребувати великої кількості додаткових обчислень, об’ємів пам’яті для зберігання дуже великих обсягів даних, покращеної швидкості передавання даних, та розвиток технологій цього напряму.

Також не потрібно забувати про методи та технології, що дозволяють зберігати, аналізувати та обробляти різноманітні, я в кращому випадку більшість доступних типів мультимедійних даних. А також таких, що дають можливість переформатовати однотипні дані з близьких типів з одного у інший, або навіть надаючих своєрідний універсальний контейнер.

Для знаходження підтвердження вищезазначених тез навіть не потрібно проводити спеціальні довгі дослідження або бути висококласним спеціалістом у даній сфері.

Будь який користувач, що навіть іноді стикався з роботою за комп’ютером безумовно потрапляв у ситуацію, коли формат бажаного відео або аудіо виявлявся несумісним з наявним програмним забезпеченням, або просто потрібна була можливість перетворити його у щось інше, просто більш зручне для використання у даний момент. Даний приклад може майже стовідсотково проілюструвати необхідність подібних речей у сучасному потоці все збільшуючоїся кількості різноманітних мультимедійних і не тільки типів даних.

І це тільки одна з проблем. Різноманітні сучасні технології, що обробляють різноманітні дані та зберігають їх мають гостру необхідність в універсальності подібних застосунків, тому їх розробники в першу чергу ламають голови над створенням складних алгоритмів та контейнері, що дають змогу реалізувати цю необхідну частину функціоналу для кожної програми.

Також досить перспективним напрямом є розширення можливостей застосування мультимедійних даних у сферах освіти та набираючого популярність 3D друку [1].

Програмне забезпечення у даних сферах повинно бути максимально універсальним та простим, і в той же час підтримувати максимально можливу кількість типів даних для підтримки свого загальнодоступного статусу.

Усі вищезазначені фактори ставлять досить складну задачу перед розробниками та зобов'язують розробляючих таке програмне забезпечення спеціалістів постійно звертати свою увагу такі компоненти свого продукту, що поскладнює їм життя.

Отже, як можна побачити з вищезазначеного, мультимедія являє собою сферу, що постійно розвивається, та в якій поки досі не знайдені універсальні рішення. З чого можна зробити висновок о доцільності даної роботи.

1.2. Аналіз аналогів

В даному розділі буде наведено різноманітні програмні застосунки, що так або інакше являють собою конкурентні засоби або аналоги розроблюваного в даному дипломному проекті програмного забезпечення.

Будуть описані їх основні принципи роботи, конкурентні переваги та недоліки, та на основі цих даних буде проведено аналіз доцільності виконання даного дипломного проекту.

1.2.1. Media Foundation

Media Foundation – це в першу чергу мультимедійний фреймворк і інтерфейс програмування застосунків, створений корпорацією Microsoft для роботи з цифровим мультимедіа під операційною системою Windows. Написаний в розрахунку на використання з C / C ++ [2].

Згідно з планами Microsoft, він призваний замінити застарілі версії, такі як наприклад DirectShow [3] або Windows Media SDK [4].

Архітектуру Media Foundation можна представити у вигляді шару управління, шару ядра та шару платформи [5]. Детальніше роздивитися цю схему можливо на рис. 1.1.

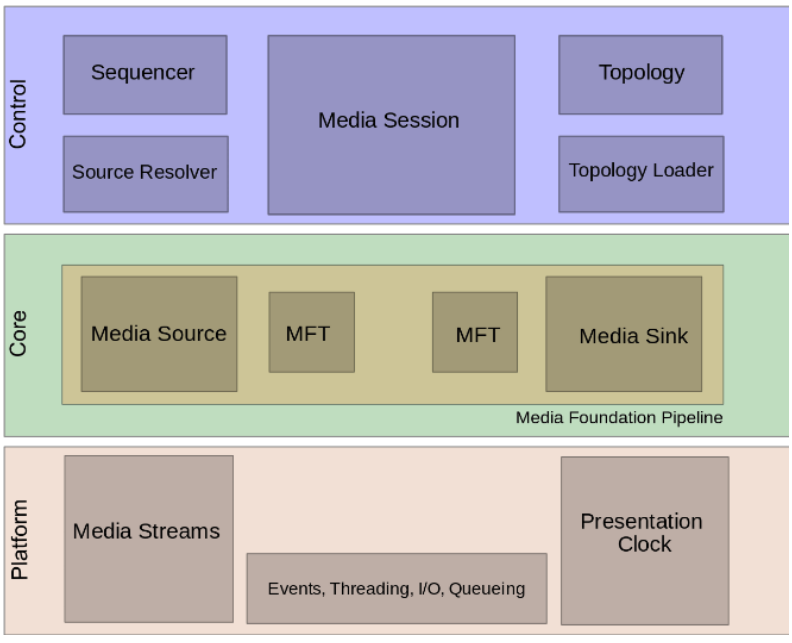


Рис. 1.1. Архітектура Media Foundation

Шар ядра містить у собі більшу частину функціональності Media Foundation. Цей шар на сам перед являє собою мультимедійний конвеєр, що складається з трьох наступних частин:

1. Media Source (об'єкт-«джерело» мультимедіа даних)
2. Media Sink (об'єкт-«приймач» оброблених даних)
3. Media Foundation Transforms (об’єкт-«трансформатор»)

Об’єкти-джерела являють собою як не парадоксально різноманітні джерела різних типів мультимедійних даних. З універсальним входом, загальним для усіх типів даних. Такі дані можуть отримуватися наприклад з файлової системи (відео та аудіо треки), з мережі Інтернет, або з зовнішніх пристроїв запису інформації.

Об’єкти-приймачі являють собою своєрідний кінцевий пункт шляху мультимедійних даних. В загалом їх можна розділити на два типи: візуалізатори та архіватори. Перший тип відтворює те що знаходилося у джерелі за допомогою пристроїв виведення (наприклад монітор, проектор, тощо), а другий просто зберігає отримані дані у файловій системі.

Основними перевагами Media Foundation над попередниками, і зокрема над DirectShow є:

1. Масштабованість для багаторозмірного контенту та контенту, який захищений DRM [6].
2. Дозволяє використовувати DirectX-акселерацію [7] відео без використання інфраструктури DirectShow.
3. Має велику сумісність з різноманітними сервісами та службами захисту контенту.
4. Media Foundation використовує Multimedia Class Scheduler Service [8]. Нову службу, яка виставляє відтворенню мультимедіа пріоритет реального часу для резервування необхідних при відтворення ресурсів. MMCSS гарантує мультимедійним додаткам пріоритетний доступ до ресурсів ЦП, який забезпечує більш точне за часом відтворення мультимедіа.

1.2.2. FFMPEG

FFmpeg – це комплекс вільного програмного забезпечення та бібліотек для різноманітних маніпуляцій з цілим спектром мультимедійних даних. Наприклад запис, відтворення, конвертація та збереження у різних форматах.

Основними перевагами даного програмного продукту є наявність цілої низки відео та аудіо кодеків та інтуітивно зрозумілий інтерфейс.

Даний фреймворк складається з таких компонентів як інструменти командного рядка та бібліотеки [9].

Також даний засіб є кросплатформним, і хоча початково він розроблений під Linux, він успішно працює під операційеі системи Windows та Mac OS [10].

1.2.3. Media Lovin' Toolkit

Media Lovin' Toolkit – це мультимедійний фреймворк з відкритим кодом, який був спеціально спроектований та розроблений для телевізійної передачі даних [11].

Він надає досить великий набір інструментів для телебачення, відеоредакторів, медіаплеєрів, транскодерів і багатьох інших типів програмного забезпечення.

Функціональність даного програмного застосунку забезпечується за допомогою асортименту готових до використання інструментів, компонентів розробки XML і легко масштабованого API на основі плагінів.

Media Lovin 'Toolkit надає API з мінімальними залежностями (POSIX і C99). А його модульна конструкція дозволяє досить легко додавати нові компоненти та інтегруватися з іншими мультимедійними бібліотеками та додатками.

За допомогою даного програмного забезпечення користувач може створювати та маніпулювати мультимедійними даними, які прив’язані до

часу. Наприклад можна привести плейлісти, комбіновані доріжки, фільтри та інше.

API даного фреймворку існує для багатьох мов програмування, таких як C ++, Java, Lua, Perl, PHP, Python, Ruby і Tcl. Також Media Lovin 'Toolkit використовує переваги багатоядерних обчислень.

Завдяки модульній конструкції даний програмний застосунок також підтримує декілька сторонніх бібліотек, таких як FFmpeg і Jack. Наприклад за допомогою FFmpeg, Media Lovin 'Toolkit здатний підтримувати майже всі аудіо та відеоформати.

1.3. Обґрунтування актуальності розробки

Оцінимо наведені технічні рішення за допомогою шкали рівня технологічної готовності [12], дана шкала схематично представлена нижче, на рис. 1.2.

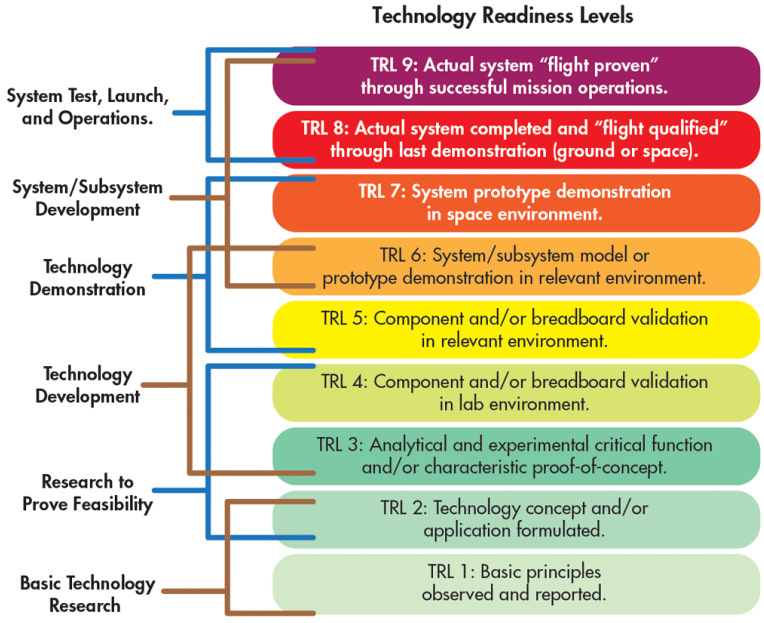


Рис. 1.2. Шкала рівня технологічної зрілості

Як можна побачити у попередніх пунктах, хоча на даний момент і існує деяка кількість фрейморків для полегшення роботи з

мультимедійними даними, які знаходяться на технологічних рівнях 7-9, усі вони являють собою тільки надбудови на вже існуючі засоби, що початково вцілому не призначені для роботи з широким спектром мультимедійних даних. А продвинуті рішення мають застосунок лише у конкретно відділеній області.

Отже, рішення створити універсальну мову, що дозволяє обробляти широкий спектр мультимедійних даних.

1.4. Висновки до розділу

У даному розділі було надано фактичні приклади та розглянуто поточний стан програмного забезпечення призначеного для роботи з мультимедійними даними.

На основі результатів проведеного аналізу можна зазначити, що не зважаючи на стрімкий розвиток апаратної частини, мультимедіа відчуває деяку недостачу в програмній частині. І завдяки цим даним можна зробити висновок, що існує необхідність у розробці програмного забезпечення для полегшення обробки мультимедійних даних з підтримкою максимальної кількості форматів збереження мультимедійних даних.

Отже, можна сказати, що розробку програмного забезпечення, що має за мету полегшення обробки мультимодальних даних для пересічного користувача можна вважати доцільною.

2. ОБҐРУНТУВАННЯ ВИБОРУ ЗАСОБІВ РЕАЛІЗАЦІЇ

В даному розділі будуть розглянуті засоби, використання яких може бути доцільним у розробці даного програмного продукту, та з яких будуть обрані найкращі інструменти.

В першу чергу розглянемо доступний набір високорівневих мов програмування, на яких буде написано ядро даного проекту. Для поліпшення процесу розробки мови повинні відповідати наступним вимогам:

1. Зрозумілий та чіткий синтаксис
2. Статична типізація
3. Поглиблений інструментарій для кращого доступу до пам’яті
4. Наявність вичерпної документації
5. Підтримка за потреби інтеграції сторонніх засобів

В даному випадку мова, яка найбільше підходить для написання даного програмного продукту буде відібрана на основі відповідності вищезазначеним критеріям та судженню автора про доцільність її використання, тому будуть проаналізовані 5 наступних мов програмування, обрані за індексом TIOBE у 2019 році [13], це: Java, C, C++, C# та Python.

2.1. Мова програмування Java

Java – це мова програмування, початково розроблювана компанією Sun Microsystems.

Основною особливістю Java-програм є перетворення написаного користувачем в спеціальний байт-код, завдяки чому такі програми можуть бути запущені у будь якому середовищі, що підтримує машину Java. Станом на 2019 рік Java є однією з найпопулярніших мов програмування [14]. Це в свою чергу є гарантією того, що для рішення існує вичерпна документація.

Перевагою виконання програми на віртуальній машині є повна незалежність виконуваного коду від програмного і апаратного середовищ, що дозволяє запускати програми на Java на будь-де, де існує працююча віртуальна машина.

Також одною з особливостей Java є стійка система безпеки, яка заснована на тому, що виконання програми на Java контролює сама віртуальна машина [15]. Тому дії, що перевищують встановлені права програми (тобто, доступ до даних які не належать програмі або підключення до іншого комп'ютера), можуть викликати негайне переривання.

Програми Java переводяться в байт-код, який виконується відповідною Java машиною. Яка в свою чергу обробляє байт-код і надсилає інструкції до апаратної частини як інтерпретатор.

Одним з основних недоліків даної віртуальної машини є суттєве зниження продуктивності. Однак ряд деяких поліпшень дещо збільшив швидкість виконання програм Java. Серед них можна виділити наступні, такі як:

1. Використання технології перекладу байт-коду в машинний код безпосередньо під час виконання програми (JIT-технологія) з можливістю збереження версій класу в машинному коді.
2. Використання специфічного для платформи коду у стандартних бібліотеках.
3. Апаратне забезпечення, що значно прискорює оброблення байт-коду (наприклад, технологію Jazelle, підтримувану деякими процесорами архітектури ARM).

Згідно з інформацією наданої сайтом wikipedia [16], для тестового набору завдань, час виконання програми в Java в середньому становить у 1.5-2 рази більше, ніж для Сі/ Сі++, варто зазначити, що іноді Java працює швидше, а іноді у сім раз повільніше.

Також одним із недоліків можна зазначити той факт, що споживання фізичної пам'яті Java в 10–30 разів перевищувало аналогічне у програми написаної на C / C++ [17].

Також може заслуговувати увагу дослідження, яке представила компанія Google, яке також вказує на те, що має місце значно нижча продуктивність і більш високе використання пам'яті для програм Java порівняно з аналогічними програмами C++.

В загалом, ідеї втілені в концепції на різних реалізаціях середовища віртуальних машин Java, надихнули багатьох ентузіастів розширити список мов, які можна використовувати для створення програм, що працюють за допомогою віртуальних машин. Ці ідеї також відображені в специфікації спільної мовної інфраструктури CLI [18], яка є основою платформи з загальною назвою Microsoft .NET [19].

2.2. Мова програмування С

Мова С – мова програмування загального призначення, розроблена як подальший розвиток мови B. Початковою ціллю розробки була у реалізації операційної системи UNIX [20], але згодом дана мова програмування була перенесена на багато інших платформ.

Конструкції цієї мови тісно пов'язані з типовими машинними інструкціями, тому вона знайшла застосування в проектах, для яких доцільно використовувати мову асемблера, тобто як в операційних системах, так і в різних прикладних програмах для багатьох пристроїв – від суперкомп'ютерів до вбудованих систем .

Мова програмування С суттєво вплинула на розвиток індустрії програмного забезпечення, і його синтаксис став основою для таких мов програмування, як C ++, C #, та Java.

Мова С була розроблена як мова системного програмування, для якої можна створити однопрохідний компілятор. Стандартна бібліотека мови

також не дуже велика. І, як наслідок, компілятори можна розробити відносно легко. Тому ця мова доступна на багатьох платформах. Крім того, незважаючи на свій низький рівень, мова орієнтована на багатоплатформність. Також програми, що відповідають стандартам мови, можуть бути скомпільовані для різних комп'ютерних архітектур.

Метою мови було полегшити написання великих програм з мінімізацією помилок у порівнянні з асемблером, дотримуючись принципів процедурного програмування, але уникаючи всього, що може призвести до додаткових накладних витрат, характерних для мов високого рівня.

Можна виділити наступні основні особливості С [21]:

1. Простий синтаксис, з якого у стандартну бібліотеку було перенесено багато основних функцій, такі як математичні функції або функції роботи з файлами.
2. Орієнтація на процедурному програмуванні.
3. Слаба статична типізація.
4. Використання препроцесора для абстрагування подібних операцій.
5. Доступ до пам'яті за допомогою вказівників.
6. Невелика кількість ключових слів.
7. Наявність покажчиків на функції і статичних змінних.
8. Області імен.

Однак у С відсутні:

1. Вкладені функції.
2. Засоби автоматичного управління пам'яттю.
3. Вбудовані засоби об'єктно-орієнтованого програмування.
4. Засоби функціонального програмування.

Деякі з відсутніх можливостей можуть бути змодельовані за допомогою вбудованих інструментів (наприклад, імітація параметрів можна виконувати за допомогою функцій setjmp і longjmp), деякі додаються за допомогою сторонніх бібліотек (наприклад, для підтримки багатозадачності та для мережевих функцій можуть використовувати pthreads, сокети тощо, є

бібліотеки для підтримки автоматичного збору сміття), частина реалізована в деяких компіляторах як розширення мови (наприклад, вкладені функції в компілятор GCC [22]).

Також існує кілька громіздкий, але цілком працездатний метод, який дозволяє реалізувати механізми ООП, засновані на фактичному поліморфізмі вказівників у мові С і підтримці вказівників на функції у цій мові. Механізми ООП реалізуються в бібліотеці GLib [23] і активно використовуються в рамках фреймворку GTK+ [24]. GLib забезпечує базовий клас GObject, здатність спадкування від одного класу і реалізацію декількох інтерфейсів.

В загалом, мова досить росповсюджена, що виступає гарантом того, що мова підримується на різних платформах та для неї існує свормована та вичерпна документація, яку зручно використовувати для розробки програмного забезпечення, а отже програмісти мають можливість досить точно зрозуміти, як саме виконуються їхні програми.

Завдяки своїй близькості до мов низького рівня програми на мові С працювали більш ефективно, аніж написані на багатьох інших мовах високого рівня.

2.3. Мова програмування С++

Мова С++ – це складена, статично типізована мова програмування загального призначення.

Підтримує такі парадигми програмування, як процедурне програмування, об'єктно-орієнтоване програмування, загальне програмування. А також має багату стандартну бібліотеку, яка включає загальні контейнери та алгоритми, регулярні вирази, підтримку багато поточності та інші функції. С ++ поєднує властивості як мов високого рівня, так і мов низького рівня. У порівнянні зі своїм попередником, мовою з іменем С, найбільшу увагу приділено підтримці об'єктно-орієнтованого і узагальненого програмування.

Мова C++ широко використовується для розробки програмного забезпечення, та є однією з найпопулярніших мов програмування. Її сфера застосування охоплює створення операційних систем, різноманітні прикладні програми, драйвери пристроїв, додатки для вбудованих систем, високопродуктивні сервери та ігри. В загалом існує багато реалізацій даної мови програмування, як вільної, так і комерційної, і для різних платформ. Також варто зазначити, що C ++ мав величезний вплив на інші мови програмування, насамперед на Java і C #.

Синтаксис C++ успадковується від C. Так як одним з основних принципів розробки була підтримка сумісності з C. Однак C ++ не є над множиною C. І хоча набір програм, які можна однаково успішно компілювати як компіляторами C, так і компіляторами C ++, досить великий, але він не включає усі можливі програми на C.

Мова C++ містить засоби розробки програмного забезпечення з контрольованою ефективністю для широкого кола завдань, від утиліт низького рівня та драйверів до дуже складних програмних пакетів.

Серед основних переваг цієї мови можна виділити наступні [25]:

1. Висока сумісність з мовою C. Тобто код C можна компілювати з мінімальними модифікаціями компілятором C ++. Зовнішній мовний інтерфейс прозорий, тому бібліотеки C можна викликати з C ++ без додаткових витрат.
2. Як наслідок попереднього пункту – обчислювальна продуктивність. Ця мова покликана дати програмісту максимальний контроль над усіма аспектами структури та порядку виконання програми. Одним з основних принципів C ++ є "ви не платите за те, що ви не використовуєте" – тобто жодна з функцій мови, які призводять до додаткових накладних витрат, не є обов'язковими для використання . Також існує можливість працювати з пам'яттю на низькому рівні.
3. Підтримка різних стилів програмування: традиційне імперативне програмування (структурований, об'єктно-орієнтований), узагальнене

програмування, функціональне програмування, генерування мета програмування.

4. Автоматичний виклик деструкторів об'єктів в належному порядку (зворотний виклик для конструкторів) спрощує і підвищує надійність керування пам'яттю та іншими ресурсами (відкриті файли, мережні підключення, з'єднання з базами даних тощо).
5. Перевантаження оператора дозволяє коротко і просторо писати вирази для користувацьких типів в природному алгебраїчному вигляді.
6. Можна контролювати константність об'єктів (модифікатори const, mutable, volatile). Використання постійних об'єктів підвищує надійність і служить підказками для оптимізації.
7. Шаблони C++ [26] дозволяють створювати загальні контейнери та алгоритми для різних типів даних. Також шаблони дозволяють виконувати розрахунки на етапі компіляції.
8. Можливість вбудовування предметно-орієнтованих мов програмування в основний код. Цей підхід використовує, наприклад, бібліотеку Boost.Spirit, яка дозволяє вказувати граматику EBNF парсерів безпосередньо в C ++ коді.
9. Доступність. Для C ++ існує величезна кількість підручників, перекладених на різні мови. І хоча мова має високий поріг вступу, але серед усіх мов такого роду він має найширші можливості.

З попередньо перерахованих пунктів випливає те, що дана мова містить у собі весь необхідний інструментарій для створення майже будь-якої програми, а її розповсюдженість є гарантом наявності вичерпної документації.

2.4. Мова програмування C#

C # – об'єктно-орієнтована мова програмування, розроблена групою інженерів Microsoft як мова розробки додатків для Microsoft .NET Framework.

Мова C # відноситься до сімейства мов з C-подібним синтаксисом і його синтаксис найбільш близький до таких мов C ++ або Java. Мова статично типізована, підтримує поліморфізм, перевантаження оператора, делегатів, атрибутів, подій, властивостей, загальних типів і методів, ітераторів та анонімних функцій з закриттями.

Мова C# багато перейняла від своїх попередників – C ++, Pascal, Modula, Smalltalk і, зокрема, Java. Тому C #, виходячи з практики їх використання, виключає деякі моделі, що виявилися проблематичними при розробці програмного забезпечення, наприклад, C #, на відміну від C ++ і деяких інших мов, не підтримує наслідування від декількох класів.

Мова C # був розроблений як мова на рівні додатків для CLR [27] і тому залежить насамперед від можливостей самого CLR. Це стосується, в першу чергу, системи типу C #, яка відображає BCL [28]. Наявність або відсутність певних виразних ознак мови диктується тим, чи може конкретна мовна особливість бути переведена у відповідні конструкції CLR. І хоча таку взаємодії і слід очікувати в майбутньому вона частково була порушена з виходом C # 3.0 [29], який є розширенням мови, що не покладається на розширення .NET.

Також C # підтримує інструменти для безпосередньою роботою з пам’яттю, але такі шматки коду мають позначатися ідентифікатором unsafe, та вимагають спеціального дозволу.

Згідно з інформацією наданою сайтом mattwarren.org, код написаний на C# у порівнянні з C/C++ є приблизно на 30% повільніший [30], за рахунок більшої різниці синтаксису від низькорівневого машинного коду який виконується комп’ютером.

І хоча існуюча документація стандартних бібліотек може відрізнятися від версії до версії, в загалом вона цілком прийнятна та добре зрозуміла для сприйняття. Що також гарантується великою кількістю розробників, що спеціалізуються на даній мові.

2.5. Мова програмування Python

Python – мова програмування загального призначення, орієнтована на підвищення продуктивності розробника і читаності коду.

Синтаксис підтримуваний ядром Python має досить малий об’єм. Однак стандартна бібліотека включає в себе велику кількість корисних додаткових функцій, це також створює необхідність досліджувати сторонні бібліотеки на наявність потрібного в даному випадку модулю.

Python підтримує структуроване, об'єктно-орієнтоване, функціональне, імперативне та аспектно-орієнтоване програмування. Основними архітектурними особливостями є динамічна типізація, автоматичне управління пам'яттю, повна інтроспекція, механізм обробки виключень, підтримка багатопотокових обчислень, високорівневі структури даних. Також підтримується поділ програм на модулі, які, у свою чергу, можуть бути об'єднані в пакети.

Python є активно розвиваючоюся мовою програмування [31], нові версії з додаванням / зміною властивостей мови виходять приблизно раз на дві з половиною роки. Мова не підлягала офіційній стандартизації, роль стандарту фактично виконувала CPython [32], розроблена під контролем автора мови.

Варто зазначити, що портування ядра Python виконано майже для всіх платформ, однак той же час, на відміну від багатьох інших портативних систем, Python підтримує технології, характерних для цієї платформи, прикладом може слугувати, Microsoft COM / DCOM [33].

Більш того, існує версія Python розроблена для Java – Jython [34], яка дозволяє інтерпретатору працювати на усіх сиситемах, які підтримують

Java, тоді як Java-класи можна використовувати безпосередньо з Python і навіть написані на Python. Також декілька розширень мають за мету забезпечити сумісніть Microsoft .NET, головною з яких є IronPython [35] і Python.Net [36].

Із зазначених вище тез випливає те, що Python хоча і залежний від встановлення свого ядра, так само як і Java, однак може виконуватися на великій кількості сучасних платформ.

Також Python є популярною мовою програмування, насамперед для рішень різних математичних алгоритмів, з цього випливає величезна кількість вичерпної технічної документації, яка майже повністю покриває описання доступних модулів.

2.6. Порівняльна оцінка

В даному пункті буде проведено порівняння мов програмування, за такими параметрами як: швидкодія, зручність синтаксису, гнучкість засобів для роботи з компіляторами та доступність навчальних засобів та документації. Оцінка буде проводитись за шкалою від 0 до 10 та є відображенням власної думки автора, результати порівняння можна переглянути у табл. 2.1.

Таблиця 2.1

Порівняльна оцінка мов програмування

	Java	C	C++	C#	Python
Швидкодія	7	10	10	8	8
Синтаксис	8	8	8	10	9
Засоби	9	5	10	8	9
Документація	10	9	10	10	10
Середня оцінка	8	8	9.5	9	9

В даному випадку вирішено віддати перевагу мові C++, тому, що дана мова поєднує у собі властивості високорівневих мов програмування з можливістю роботи з пам’яттю, а також має велику кількість різноманітних матеріалів на дану тему.

Отже, беручи до уваги власний досвід у програмуванні та вищезазначені аргументи можна вважати доцільним обрати мовою реалізації C++.

3. РОЗРОБЛЕННЯ АЛГОРИТМІВ ТА ПРОГРАМНИХ МОДУЛІВ

3.1. Загальні положення проекту

Основну ідею проекту можна сформулювати наступним чином, на вхід до програми у загальному випадку подається потік символів, у більш конкретному випадку це можна представити як будь який файл у файловій системі.

За допомогою лесичного аналізу з вхідного потоку виокремлюються окремі лексичні конструкції – лексеми.

Оброблена послідовність передається до модулю синтаксичного та семантичного аналізу, продуктом праці якого є абстрактне синтаксичне дерево.

Отже, на основі вищезазначеного опису доцільно буде виокремити наступні три основні модулі даного програмного забезпечення:

1. Лексер;
2. Парсер;
3. Інтерпретатор.

Безумовно це далеко не всі компоненти розробленого програмного забезпечення, однак інші, за виключенням контейнеру для абстрактного синтаксичного дерева, слугують допоміжними інструментами, що дозволяють відлагоджувати та контролювати працездатність програмного забезпечення.

Більш детально загально описану вище структуру транслятору можна розглянути на рис. 3.1.

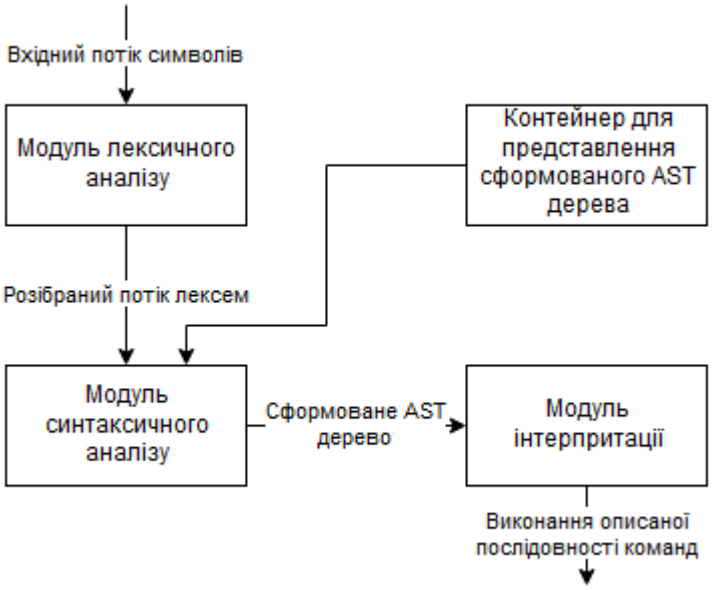


Рис. 3.1. Узагальнена схема роботи транслятору

Більш детально ці три основні модулі будуть розглянуті у наступних пунктах даного розділу дипломного проекту.

3.2. Короткий опис граматики

В даному розділі будуть розглянуті основні положення граматики мови, на основі якої розроблювався даний дипломний проект.

Основний опис граматики мови у форматі EBNF можна детальніше розглянути у додатках до пояснювальної записки, тому тут буде наведено лише основні положення.

Код виконуваної програми розпочинається кодовим словом program, після якого слідує ім’я, та фігурними дужками визначається простір виконуваної програми.

Тіло програми складається з дев’яти ключових розділів, загальну схему яких, у свою чергу можна представити наступним чином.

Лістинг 3.1. Загальна схема розділів програми

```
'Program', name, '{',
      libraries section,
      handlers section,
      renderers section,
      sources section,
      sets section,
      elements section,
      tuples section,
      aggregates section,
      actions section,
'}';
```

Секція бібліотек дозволяє оголосити список імпортованих бібліотек.

Наступні дві секції – секція обробників та секція рендерів дозволяють виокремити дані компоненти з оголошеної бібліотеки для їх подальшого, більш зручного використання.

Секція джерел складається з переліку оголошень звернень до сторонніх ресурсів.

Секція наборів дозволяє оголосити або імпортувати з якоїсь бібліотеки свій власний тип.

Секція елементів дозволяє оголошувати зміні, яким буде представлений ідентифікатор для подальшого використовування у коді програми.

Секція кортежів має за мету оголошувати перелік змінних типу кортеж, найбільш близькою аналогією буде масив різних значень, або структура.

Секція ж агрегатів у свою чергу дозволяє оголошувати кортеж кортежів.

Секція дій містить у собі реалізацію логіки написаної програми і складається з будь якої кількості та послідовності дій, які передбачені граматикою мови.

3.3. Лексер

Лексер – модуль обробки вхідної послідовності символів, який виконує процес аналітичного розбору вхідної послідовності символів шо подається на нього з метою отримання послідовності лексем (токенів).

Сам лексичний аналіз проводиться з точки зору певного формального набору правил, яким в даному випадку виступає розроблена граматика мови ASAMPL. Вона задає певний набір лексем які можіть зустрітися у вхідній послідовності символів.

Розпізнавання лексем у контексті обраної граматики відбувається шляхом їх ідентифікації серед того набору лексем які надаються граматикою мови.,

При чому будь яка послідовність символів вхідного потоку, яка згідно до граматики не може бути ідентифікована як та, що належить до цієї граматики розглядається як непізнана лексема, або інакше кажучи лексема-помилка.

Кожен виокремлену лексему у даній реалізації можна представити у вигляді об’єкту або структури, що містить ідентифікатор лексеми, її значення, та її положення у вхідній послідовності символів, для кращого оброблення помилок.

Мета роботи даного модуля складається у тому, щоб на виході дати підготовлену послідовність лексем для наступної частини даного програмного продукту, і тим самим позбавити її від лексичних подробиць у контекстно-вільній граматиці.

Узагальнену схему роботи даного модулю можна побачити на рис. 3.2.

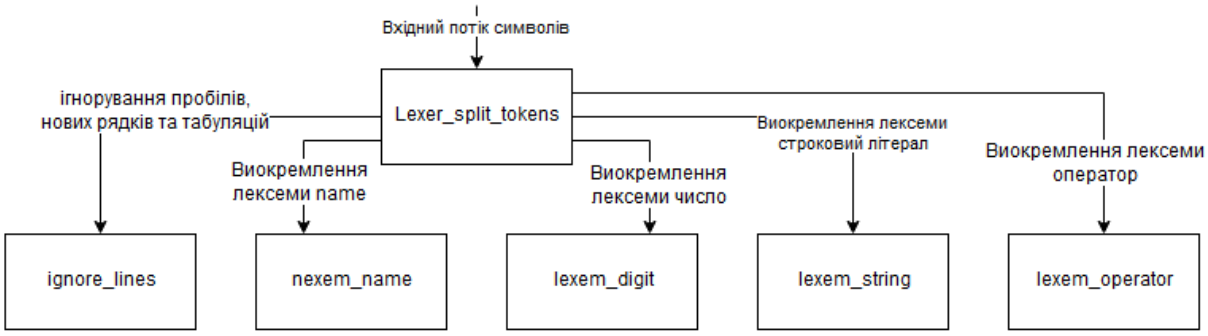


Рис. 3.2. Узагальнена схема роботи модуля лексичного аналізу

Детальніше вищевказану схему роботи модуля лексичного аналізу можна розписати наступним чином:

У основну функцію:

```
int Lexer_splitTokens(std::fstream* fs, std::vector<Lexem>* sequence)
```

За допомогою адреси розташування у файловій системі передається файловий потік та посилання на поки, що порожній список лексем, зчитування файлового потоку, його оброблення та повернення сформованого списку і є, як це зазначалося вище, основною задачею даного модулю програми.

Основним компонентом даної функції є цикл, який по черзі намагається зчитати символи з вхідного файлового потоку і намагається додати їх до списку лексем. Працює даний цикл доти, доки наступний символ файлового потоку не буде являти собою EOF. Після знаходження символу кінця файлу цикл завершує своє виконання та функція повертає щойно створений список токенів для подальшого виконання коду програми.

Детальніше роботу вищезазначеного циклу можна описати наступним чином:

Кожну ітерацію цикл спочатку виділяє наступний по черзі символ з файлового потоку, але не зчитує його.

Також створюється екземпляр класу лексеми яку необхідно буде додати до списку лексем.

На початку кожної ітерації здійснюється перевірка на наявність не несучих інформацію символів, таких як пробіл або переривання строки.

Перевірка наявності символів, не несучих інформації відбувається у наступній функції:

```
inline static int IgnoreTabNewlinesAndSpaces(std::fstream *fs)
```

У разі наявності таких символів вони циклічно ігноруються доти, доки їх можливо ігнорувати.

Наступним кроком за допомогою заздалегіть прописаних макросів по черзі перевіряються наступні умови:

1. IS_NAME_START – дана перевірка має за мету зрозуміти, чи може наступний символ бути початком ідентифікатора, і полягає в тому, чи являється наступний виділений символ літерою латинського алфавіту або символом ‘_’.
2. IS_DIGIT – дана перевірка має за мету зрозуміти, чи може наступний символ бути початком представлення числа, і полягає в тому, чи являється наступний виділений символ цифрою.
3. IS_STRING_LITERAL_START – дана перевірка має за мету зрозуміти, чи може наступний символ бути початком представлення строкового літералу, перевірка, і полягає в тому, чи являється наступний виділений символ одним з '\" або '\'".

Якщо хоча б одна з них проходить, то до неї визивається одна з відповідних функцій:

```
inline static int LexemName(std::fstream *fs, Lexem *lexem_to_add)
inline static int LexemDigit(std::fstream *fs, Lexem *lexem_to_add)
inline static int LexemString(std::fstream *fs, Lexem * lexem_to_add)
```

У тому випадку, коли жодна з перевірок не пройшла, визивається наступна функція:

```
inline static int LexemOperator(std::fstream *fs, Lexem *lexem_to_add)
```

Суть роботи вищезазначених функцій можна описати наступним чином.

Функція LexemName зчитує файловий потік доти, доки наступний символ являє собою літеру, незалежно від регістру, або будь-яке число.

Якщо наступний символ не відповідає вищезазначеній умові подальше зчитування символів завершується, і виділена лексема перевіряється на збіг з одним із ключових слів програми, якщо словник ключових слів, сформований відповідно до представленої граматики мови, містить дане ключове слово, лексемі присвоюється ідентифікатор знайденого ключового слова, інакше присвоюється LexemType_NAME.

Функція LexemDigit зчитує файловий потік доти, доки наступний символ являє собою число, або десятковий розділювач. Якщо наступний символ не відповідає вищезазначеній умові подальше зчитування символів завершується, а виділеній лексемі присвоює тип NUMBER, не зважаючи, чи був зчитаний розділювач.

Функція LexemStringLiteral LexemDigit зчитує файловий потік доти, доки наступний символ не стане являти собою символ '\" або '\'", в залежності від того, яким був відкритий строковий літерал. Якщо наступний символ не відповідає вищезазначеній умові подальше зчитування символів завершується, а виділеній лексемі присвоює тип STRING.

Функція LexemOperator на відміну від попередніх трьох не намагається циклічно зчитати файловий потік, а відразу перевіряє, чи являється наступний символ одним з базових символів граматики. І якщо це так лексемі буде присвоєно відповідний тип. При такій перевірці може виникнути ситуація коли символ є лише частиною виразу, хорошим прикладом може слугувати логічна операція більше або дорівнює, яка складається з двох символів, відповідно “>” та “=”. Для виключення таких випадків функція також перевіряє наступний символ, і якщо він підпадає під наближений до цього випадок формує лексему вже з двох символів.

Так як дана функція визивається якщо жодна з попередніх умов не пройшла, в неї також вбудований обробник помилок. Якщо у вхідній послідовності символів зустрінеться такий, який не міститься у базових

правилах граматики дана функція поверне “пошкоджену” лексему з поміткою помилки.

Після завершення роботи основна функція модулю повертає код завершення та заповнений список лексем, сформований для його подальшого розбору наступним модулем, з метою перетворення у абстрактне синтаксичне дерево.

3.4. Парсер

Парсер, або синтаксичного аналізатор, являється модулем, головною задачею якого є процес співставлення лінійної послідовності лексем формальної мови, в даному випадку мови ASAMPL, з її формальною граматикою.

У процесі синтаксичного аналізу тексту вхідна лінійна послідовність символів перетворюється у структуру даних, абстрактне синтаксичне дерево, яке наглядно відображає синтаксичну структуру вхідної послідовності, і, яка, як наслідок, відмінно підходить для подальшої програмної обробки у трансляторі.

Результатом цієї роботи буде являтися сформоване абстрактне синтаксичне дерево, структуру якого буде детальніше описано у розділі нижче.

В даній частині цього розділу буде детально описано основні принципи роботи та алгоритми даного модулю:

Отже, основним принципом даного синтаксичного аналізатору є розбір правил, реалізований за методом рекурсивного спуску [37], тобто за рахунок взаємного виклику функцій, де кожна функція відповідає одному з правил граматики.

Застосовані правила послідовно, зліва-направо поглинають лексеми, отримані від лексичного аналізатора.

За рахунок поглинання та оброблення лексем відбувається формування з них абстрактного синтаксичного дерева, яке потім буде інтерпритовано у виконуваний код.

Тепер детальніше розглянемо алгоритм роботи даного модуля, основною функцією в даному випадку є функція побудови абстрактного синтаксичного дерева:

```
Tree* parser_buid_tree(std::vector<Lexem>* lexem_sequence);
```

На вхід якої подається послідовність лексем отримана на попередньому етапі оброблення вхідних даних, а на виході отримується сформоване абстрактне синтаксичне дерево.

В даній функції заповнюються поля класу Parser, який містить у собі посилання на вхідний список лексем, ітератор для їх почергового перебору та строку для фіксування помилок, які можливо будуть формуватися при перетворені у абстрактне синтаксичне дерево. Таким чином цей клас потрібен для оброблення вхідної послідовності лексем та для формування з них абстрактного синтаксичного дерева.

Також основна функція відповідальна за оброблення помилок при формуванні абстрактоного синтаксичного дерева, тому у випадку, якщо в процесі обробленні послідовності лексем при створенні абстрактного синтаксичного дерева в класі парсер буде зафіксована помилка, то дана функція замість сформованого абстрактоного синтаксичного дерева поверне NULL.

Розбір виконується рекурсивно, тому у тілі основної функції крім створення класу парсеру та обробки повертаємого значення викликається ще тільки одна функція:

```
static Tree * program(Parser * parser)
```

Цю функцію можна назвати кореневою, в тому сенсі шо в ній по черзі викликаються обробники для кожного з так званих блоків програми, в яких будуть ініціалізуватися змінні для подальшого використання та описуватися дій програми.

Лістинг 3.2. Перелік функцій обробки розділів

```
static Tree * libraries_section(Parser * parser);
static Tree * handlers_section(Parser * parser);
static Tree * renderers_section(Parser * parser);
static Tree * sources_section(Parser * parser);
static Tree * sets_section(Parser * parser);
static Tree * elements_section(Parser * parser);
static Tree * tuples_section(Parser * parser);
static Tree * aggregates_section(Parser * parser);
static Tree * actions_section(Parser * parser);
```

Якщо помилок у результаті виконанні цих функцій не відбулося, то в такому випадку повернеться готове абстрактне синтаксичне дерево, готове для подальшого виконання, у іншому ж випадку дерево буде не повним, і невиконуваним, тому замість нього головна функція поверне NULL, та відобразить код помилки, який містить у собі інформацію про рядок, в якому вона сталася.

Тепер більш детальніше розглянемо саму схему функціонування синтаксичного розбору послідовності лексем.

В даному алгоритмі можна виділити наступні функції, що не представляють собою реалізацію одного з правил формальної граматики мови ASAMPL:

```
static Tree * accept(Parser * parser, LexemType Lexem);
static Tree * expect(Parser * parser, LexemType Lexem);
```

Функції Accept та Expect, дані функції розглядаються разом бо мають багато залежностей в реалізації.

Функція Accept, як можна судити з назви намагається зчитати наступний елемент у послідовності лексем, і співставити його тип з переданим до неї типом. Якщо типи співпадають функція успішно зчитує елемент, в інакшому випадку ні.

Expect же в свою чергу у собі викликає Accept, при його завершенні дана функція не просто перевіряє, чи являється наступна лексема конкретного типу, але ще й вимагає, щоб вона була конкретного типу

У іншому ж випадку дана функція вертає помилку та пише її у парсер, для звіту про те місце, на якому сталася помилка.

```
static bool ebnf_sequence(Parser * parser, Tree * node_to_fill, GrammarRule rule)
```

Функція Sequence, як можна побачити з назви намагається циклічно зчитати передане в неї в якості аргументу правило будь-яку кількість разів.

Використовується дана функція для додавання будь якої кількості дітей до тих листків дерева, які відповідають за ініціалізацію змінних або за виконання прописаних дій.

Якщо ж зчитування пройшло без казусів дана функція завершується, у іншому ж випадку повертає помилку у парсер та також завершується.

```
static Tree * ebnf_one_of(Parser * parser, GrammarRule rules[], size_t length)
```

Функція one_of намагається зчитати хоча б одне з правил, яке було передано у неї в якості елементів масиву.

Якщо це можливо вона повертає результат виконання цього правила як ще одну гілку абстрактного синтаксичного дерева та на цьому її виконання завершується.

У випадку коли під час виконання правила виникла помилка, або жодне зі списку переданих правил не вдалося застосувати до наступної послідовності лексем повертається NULL.

```
static Tree * ebnf_one_of_lexem(Parser * parser, LexemType types[], size_t length)
```

Функція one_of_lexem намагається зчитати хоча б одну з лексем, які були передані у неї в якості елементів масиву.

Якщо це можливо вона повертає результат ноду з типом обробленої лексеми як новий листок абстрактного синтаксичного дерева та на цьому її виконання завершується.

У випадку, коли жодну зі списку переданих лексем зчитати не вдалося повертається NULL.

```
static Tree * ebnf_ap_main_rule(Parser * parser, GrammarRule next, GrammarRule ap)
```

Функція Ap Main Rule, породження лівої рекурсії [38] перевіряє чи застосоване наступне за пріорітетом правило, і якщо це істина перевіряє чи застосовано правило з апострофом.

```
static Tree * ebnf_ap_recursive_rule(Parser * parser, LexemType types[], size_t typesLen, GrammarRule next, GrammarRule ap)
```

Функція Ap Recursive Rule, перевіряє, чи знайдені передані токени, і якщо вони застосовані чи продовжується правило рекурсивно.

Загалом, цей список можна вважати повним списком правил, які не походять з правил формальної граматики наданої мовою ASAMPL, тому з них як з компонентів, а особливо з функцій Accept та Expect, формуються усі інші правила, які є прямим породженням з формального опису граматики мови ASAMPL.

Перелік всіх інших функцій буде дуже довгим та інформативним тому краще буде описати перелік правил, який є справедливим для кожної функції яка являє собою пряме породження від правил формального опису граматики мови ASAMPL.

Кожна функція, яка являє собою пряме породження від правил формального опису граматики мови ASAMPL і не є кінцевою містить у собі виклики одної з базових функцій та, виклики мінімум одної з таких же функцій.

Кінцеві функції не містять у собі виклики подібних функцій, тому не поглиблюють рекурсію і повертають листок який є кінцем розбору даної гілки.

Приблизну схему залежності функцій, породжених формальним описом граматики мови ASAMPL можна побачити на рис. 3.3.

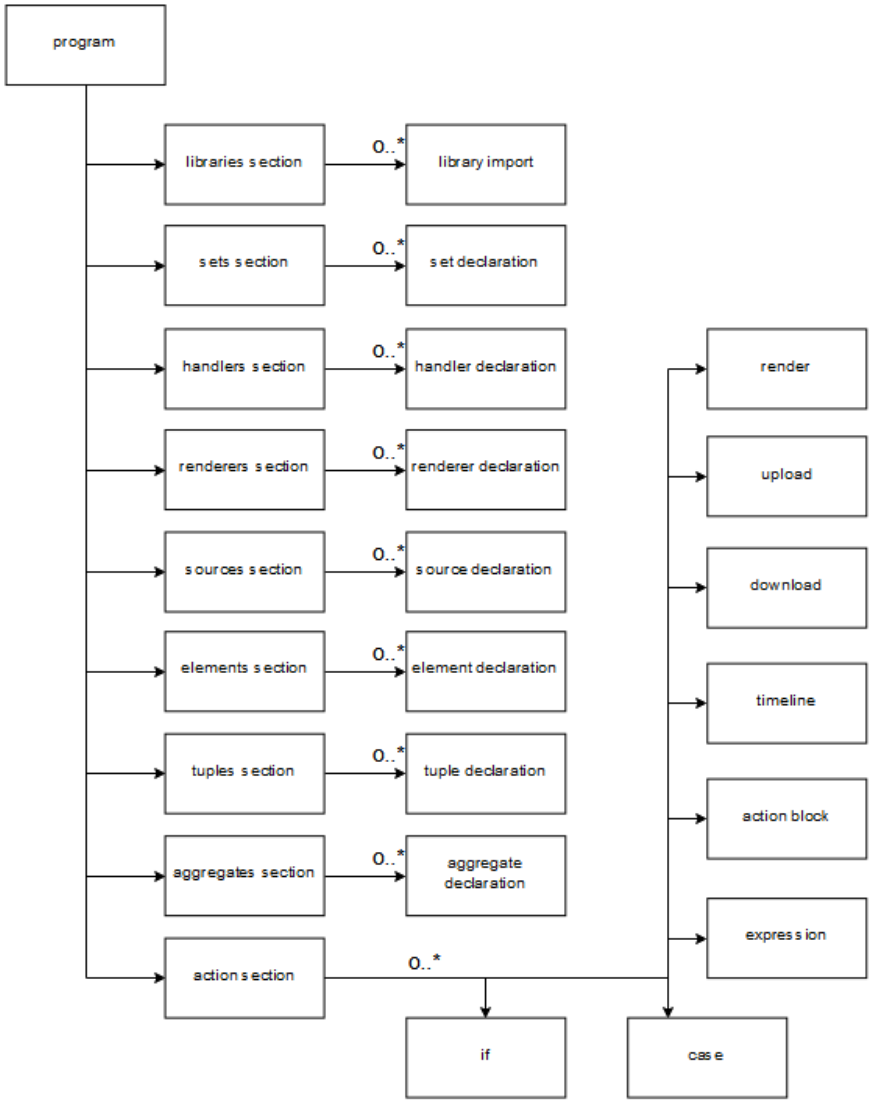


Рис. 3.3. Узагальнена схема залежностей граматичних функцій

3.5. Побудова AST

Функція побудови абстрактного синтаксичного дерева також являється частиною рекурсивного обходу наданої послідовності лексем, про який детально розписано у попередньому пункті даного розділу, але існує декілька важливих моментів про принцип роботи яких буде розказано в даному розділі.

В першу чергу слід зазначити, що не всі пройдені рекурсивно функції формують власний листок абстрактного синтаксичного дерева, особливо це стосується формування дерева для арифметичних та логічних виразів, але про них буде розписано трохи пізніше.

Першим кроком у будуванні абстрактного синтаксичного дерева є створення у батьківського листку програми, від якого в свою чергу будуть наслідуватися абсолютно усі інші листки дерева, за це відповідає функція program.

В свою чергу після цього задля полегшення подальшої обробки, що буде виконана у класі інтерпретатору створюються допоміжні листки, кожен з яких відповідає за власну частину програми. Цим займаються функції з вищезазначеного у попередньому пункті переліку обробників окремих частин програми.

Для кожного з допоміжних листків зчитується довільну кількість разів, тобто нуль або більше, те правило, або сукупність правил які можуть, згідно з формального опису граматики мови ASAMPL зустрічатися як діти цього листку.

Усі листи, крім останнього відповідальні за ініціалізацію змінних та містять лише одне правило, яке може зустрічатися довільну кількість разів. Узагальнений вигляд цього правила можна представити наступним чином:

```
element declaration = Name, assign, data, ";";
```

Де в свою чергу `data` це або якийсь арифметичний або логічний вираз, або строковий літерал, або ідентифікатор, або ініціалізація масиву.

Останній ж листок, actions, являє собою послідовність довільного розміру, яка може складатися з усіх можливих дій, а значить прописаних у граматиці мови ASAMPL, дій, виконання яких і являє собою основну ціль програми.

Для кожної дії формується окремий відповідний листок абстрактного синтаксичного дерева, які прикріплюються один до одного, або до батьківського листку дій.

В залежності від правил формування даних листків, та для полегшення інтерпретації уся необхідна для них інформація також записується у відповідні дочірні листки.

Для кожної з можливих дій порядок розташування дітей абстрактного синтаксичного дерева є сталий і може буди досліджений у додатку граматики.

Формування ж листку будь якого виразу формуються у відповідності до загальноприйнятного пріоритету операцій, який можна наглядно побачити у табл. 3.1.

Таблиця 3.1

Таблиця пріорітетів операцій

Пріоритет	Група операцій	Операції
1	Первинні	( ) Усі дії у дужках
2	Унарні	! + -
3	Мультиплікативні	* /
4	Адитивні	+ -
5	Відношення	< <= > >=
6	Порівняння	== !=
7	Логічне ТА	&&
8	Логічне АБО	||
9	Присвоєння	=

3.6. Структура AST

Контейнер для збереження будуючогося у модулі синтаксичного аналізу абстрактного синтаксичного дерева з трьох наступних компонентів, які перераховані нижче:

1. Перелік усіх можливих для даної граматики типів листків абстрактного синтаксичного дерева, що реалізований у вигляді нумератору під назвою AstNodeType.
2. Клас представлення самого листку абстрактного синтаксичного дерева під назвою AstNode, який у свою чергу складається з поля type, типу ноди представленого вище описаним нумератором та поля value, в якому зберігається інформація, що належала даній ноді.
3. Клас представлення самого дерева під назвою Tree, який складається з представлення листку абстрактного синтаксичного дерева та зі списку, який вказує на усіх його нащадків.

Даний контейнер створюється та заповнюється у класі синтаксичного аналізу списку лексем, або парсером, та передається до оброблення у клас інтерпретатору.

Діаграму класів абстрактного синтаксичного дерева можна детальніше розглянути на рис. 3.4.

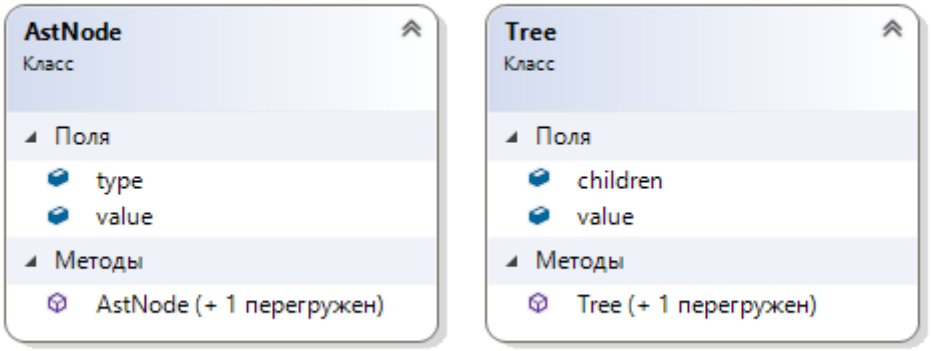


Рис. 3.4. Діаграма класів абстрактного синтаксичного дерева

3.7. Транслятор

Модуль транслятор виконує по операторну обробку вхідної послідовності команд, яка має бути представлена у вигляді абстрактного синтаксичного дерева.

Дана частина розробленого програмного продукту призвана реалізувати виконання команд, що описані о формальному представленні граматики мови ASAMPL.

Основними компонентами даного модулю можна назвати наступні його частини:

Клас Program, основна мета якого зберігати усі необхідні для виконання програми змінні та реалізовувати зручний до них доступ з будь-якої частини коду, а також зберігати, обробляти та видавати помилки при виконанні програми, написаної з використанням формальної граматики мови ASAMPL.

Клас Value, головна мета якого представляти контейнер для змінної будь якого типу, у зв’язку з реалізацією динамічної типізації для даної мови програмування.

Для розпізнавання конкретного типу з метою коректної обробки тих даних, що в момент обробки знаходяться у контейнері реалізовано нумератор з переліком усіх основних типів.

Детальніше діаграму класів інтерпретатору можна детальніше розглянути на рис. 3.5.

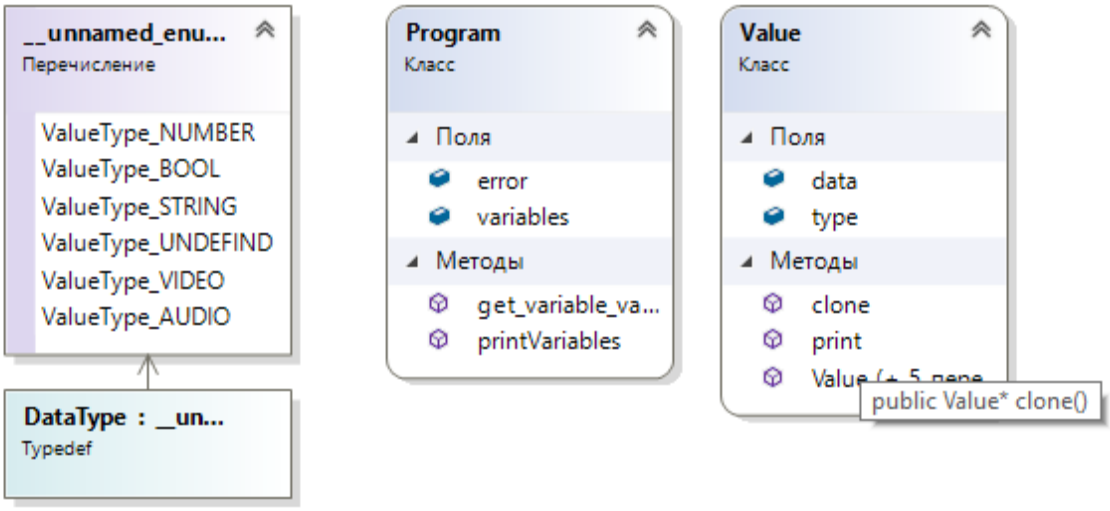


Рис. 3.5. Діаграма класів інтерпретатору

Більш детально принцип роботи даного модуля можна описати наступним чином:

У головну функцію даного модуля:

```
int execute(Tree * astTree)
```

Передається створене у попередньому модулі абстрактне синтаксичне дерево з вказівником на його вершину, якою в свою чергу виступає листок типу program.

У тілі даної функції міститься цикл, який по черзі перебирає усі дочірні листки кореня, які в свою чергу являють собою послідовність розділів програми.

За допомогою оператору вибору для кожного окремого прописаного розділу у циклі функції викликається відповідний його обробник, який в залежності від реалізації намагається виконати ті чи інші прописані у розділі дії .

Для розділів з ініціалізацією різноманітних змінних це відповідно ініціалізація змінних с виразів або ідентифікаторів, а для розділу який відповідає за виконання дій це в свою чергу їх виконання з метою отримання певного результату.

При виконанні розділу з ініціалізацією змінної інтерпретатор спочатку виконує поміщений у праву частину від оператору присвоєння вираз. Якщо виконання виразу пройшло коректно, інтерпретатор створює змінну з унікальним, записаним користувачем ідентифікатором, та записує її у словник, який міститься у класі програм у вигляді ключ-значення. Ключем виступає створений користувачем ідентифікатор, а значенням являється клас з типом Value, який був створений у результаті попереднього виконання інтерпретатором виразу.

У випадку якщо ідентифікатор, який прописав користувач для змінної не може бути унікальним ключем, програма видасть помилку – "Duplicate variable id".

Після завершення виконання модулів, що відповідають за реалізацію оголошення змінних останнім у черзі виконується модуль послідовного виконання дій.

У даній частині програмного забезпечення реалізовано виконання таких наступних дій.

```
void execute_block(Program * program, Tree * blockTreeNode)
```

Функція, що оброблює виконання листку абстрактного синтаксичного дерева, який позначений як блок.

Фактично даний листок означає, що посеред операторних дужок міститься будь яка послідовність команд, і відповідно суть даної функції в тому, щоби по черзі обробити всі ці команди.

У випадку, якщо при виконанні якоїсь команди виникла помилка дана функція достроково завершує свою роботу.

```
void execute_if(Program * program, Tree * node)
```

Функція, реалізує стандартну для всіх мов програмування функцію оператора вибору.

Листок абстрактного синтаксичного дерева, який позначає дану ноду містить у собі дві наступні дитини:

Логічний вираз – умова виконання послідовності команд у операторних дужках.

Послідовність команд – у тому випадку, коли логічний вираз є істиною інтерпретатор виконує дану послідовність команд.

Також можливий ще третій нащадок – послідовність команд після ключового слова else.

Дане ключове слово не вимагається граматикою і тому може не існувати, але якщо воно існує, тоді у тому випадку коли логічний вираз є неправдою інтерпретатор виконує послідовність команд яка знаходиться між операторними дужками цього ключового слова.

```
void execute_timeline(Program * program, Tree * node)
```

Функція, яка реалізує циклічне виконання послідовності дій, доти, доки виконується умова пов’язана з часом.

Листок абстрактного синтаксичного дерева, який позначає дану ноду містить у собі дві наступні дитини:

Умова, що буває трьох наступних типів:

1. Timeline As;
2. Timeline Until;
3. Timeline Time : Time : Time.

Timline As позначає собою той варіант, в якому після ключового слова As наступною лексемою слідує або строковий літерал, який представляє собою час, або відповідна змінна.

Дана умова працює доти, доки не закінчиться вказаний в ній відрізок часу.

Timline Until позначає собою той варіант, в якому після ключового слова Until наступною лексемою слідує або строковий літерал, який представляє собою час, або відповідна змінна.

Дана умова працює доти, доки не наступить вказаний в ній відрізок часу.

Timline Time : Time : Time позначає собою той варіант, в якому умову можна задати трьома наступними значеннями:

1. Час початку виконання роботи.
2. Час закінчення виконання роботи.
3. Крок, з яким буде виконуватися тіло.

Умовою валідності поданих на вхід даної умови даних є те час початку виконання роботи повинен бути меншим часу завершення виконання роботи, інакше буде помилка.

Послідовність команд – до тих пір, поки умова є істиною інтерпретатор виконує дану послідовність команд.

Для реалізації трьох наступних функцій була використана бібліотека opencv [39], у зв'язку з тим що ще не існує допоміжних бібліотек для обробки мультимедійних даних, що були написані спеціально для даної мови програмування.

Тому цей пункт є одним з основних пунктів подальшого розвитку даного програмного засобу.

```
void execute_download(Program * program, Tree * node)
```

Листок абстрактного синтаксичного дерева, який позначає дану ноду містить у собі дві наступні дитини:

Ім'я змінної – у цю змінну буде записано значення відкритого файлу, функція створить таку змінну із заданим ідентифікатором то додасть її список змінних.

Адреса до джерела – може бути представлена у вигляді строкового літералу, або його аналогу у вигляді змінної який вказує на положення блоку у файловій системі комп'ютеру.

Функція робить спробу відкрити відеофайл за даною адресою, у успішному випадку до зміних додасться його представлення, у іншому ж випадку відобразиться помилка.

```
void execute_upload(Program * program, Tree * node)
```

Листок абстрактного синтаксичного дерева, який позначає дану ноду містить у собі дві наступні дитини:

Ім’я змінної – змінна типу відеофайлу, яка буде записана за вказаною адресою.

Адреса до отримувача – може бути представлена у вигляді строкового літералу, або його аналогу у вигляді змінної який вказує на положення файлу у файловій системі комп’ютеру.

Функція перевіряє, чи є зміна типу videofile, якщо ні – повертає помилку, якщо так записує файл за вказаним шляхом.

```
void execute_render(Program * program, Tree * node)
```

Листок абстрактного синтаксичного дерева, який позначає дану ноду містить у наступну дитину:

Ім’я змінної – змінна типу відеофайлу, наступний фрейм якого буде відображений.

Відкриває додаткове вікно, у якому відображає поточний фрейм відеофайлу.

У випадку успішного завершення виконання роботи програми процес завершиться з кодом 0, який символізує, що усе в порядку. В супротивному ж випадку програма виконає переривання своєї роботи з вказівником на причину аварійного завершення роботи і поверне мінус один.

4. АНАЛІЗ РОЗРОБЛЕНОГО КОМПІЛЯТОРА МОВИ ASAMPL

Результатом даного проекту є компілятор обробки мультимедійних даних ASAMPL.

4.1. Розгляд практичного застосування

У якості прикладу розглянемо текст програми, який буде використовуватися як тестовий, для демонстрацій можливостей створеного компілятору.

Завдання буде полягати в наступному, при виконанні деяких встановлених користувачем умов виконати зчитування відеофайлу з вказаного джерела.

Потім, за показати певний період часу показати його частину, період часу також встановлюється користувачем.

Після цього записати відкритий файл у інше місце у файловій системі комп'ютера.

Для виконання описаної вище задачі нам знадобляться такі наступні змінні:

videoIn – посилання на відеофайл у файловій системі комп’ютера, який ми збираємося обробити.

videoOut – шлях до місця у файловій системі, до якого ми збираємося записати вхідний файл.

testNumber – тестова змінна типу число для демонстрації коректної роботи обробника арифметичних на логічних виразів розробленого програмного застосунку.

Time – змінна яка буде представляти час, протягом якого буде відтворюватися відео та зявдяки якій буде продемонстровано роботу функції timeline.

Отже, для демонстрації оголошуємо перші дві змінні у розділі Sources нашої програми, третю та четверту змінні оголошуємо у розділі Elements як одиничні значення.

Змінним, які є посиланням на джерела вказуємо абсолютні шляхи у файловій системі.

Змінній testNumber вказуємо за значення будь яке число, або навіть арифметичний вираз.

Зміну Time вказуємо як строку у загальноприйнятому форматі Години:Хвилини:Секунди.

Для демонстрації корректої роботи арифметичних операцій та заодно логічних змінних використаємо оператор IF, в логічному виразі якого порівняємо значення, яке зберігається в ініціалізованій змінній та арифметичний вираз.

У якості доказу корректності виконання помістимо завантажуючий, відтворюючий та зберігаючий відео шматок коду у середину операторних дужок оператору IF.

Для завантаження відеофайлу з джерела, вказаного вище використаємо download, першом аргументом якої буде слугувати ім’я змінної яку програма буде асоціювати з відкритим відеофайлом, а другим аргументом стане змінна videoIn, яка є строкою, що містить шлях у файовій системі до пострібного нам відеофайлу.

Для відтворення відеофайлу потрібно використати такі дві наступні функції:

Render – для покадрового відтворення попередньо відкритого відеофайлу. Единим агрументом в двному випадку буде ідентифікатор відкритого вище файлу.

Timeline – для циклічного рендерингу протягом певного періоду часу, попередньо заданого в змінній time. В даному випадку ми будемо використовувати варіант Timeline as Time.

Після часткового відтворення файл необхідно записати, для цього будемо використовувати команду upload. Агрументами якої будуть змінна яка зберігає фідкритий аудиофайл та посилання на місце у файловій системі до якого буде записано даний файл.

4.2. Порівняння з існуючими засобами розроблення

В даному розділі будуть розглянуті та порівняні основні характеристики розроблюваної програми у порівнянні з існуючими засобами розробки для обробки мультимедійних даних.

Найбільш наглядне порівняння можна зробити, розглянувши приклад з тестового набору команд, що реалізує основні можливості даного програмного продукту. Програма відкриває відео файл, та покадрово відтворює його на проміжку часу який зазначений у інінціалізованій вище змінній. Для порівняння буде використана бібліотека openCV, у її реалізації для мови C++, як наглядний приклад існуючого програмно забезпечення. Тексти обох програм можна переглянути у додатках.

Як наглядно видно, програма розроблена на основі транслятору ASAMPL потребує значно менше строчок коду для описання одних і тих же дій. В основному через те, шо реалізація інтерпретатору приховує деякі функції, обробку яких не обов’язково виносити на реалізацю до програміста.

У наслідку випливає те, що спеціалізований код для оброблення мультимедійних даних написаний мовою ASAMPL більше ніж у три рази коротший, аніж еквівалентний код написаний мовою C++ з використанням бібліотеки openCV.

Як висновок до даного порівняння можна зазначити той факт, що мова ASAMPL незважаючи на громіздкі конструкції ініціалізації змінних загалом більш легка у використонні для кінцевого користувача.

4.3. Оцінка продуктивності компілятора

Продуктивність фінальної версії реалізованого компілятору мови ASAMPL була оцінена на основі двох наступних характеристик: час виконання та порівняльна кількість строчок коду для оброблення однієї послідовності дій мовою ASAMPL, та доступних з мультимедійних бібліотек методів.

Отримані результати для компілятора ASAMPL порівнювалися з результатами отриманими для компілятора G++ при розборі еквівалентних програм мовою C++. Порівняння проводилося для двох наборів тестового коду мови ASAMPL та його аналогу, написаному на мові C++.

Таблиця 4.1

Розмір виконуваного коду для обробки відеофайлів в рядках

	ASAMPL	G++
Тестовий набір №1	11	21
Тестовий набір №2	18	27

Таблиця 4.2

Середній час виконання програми у мілісекундах

	ASAMPL	G++
Тестовий набір №1	3206	3063
Тестовий набір №2	3821	3427

Порівняння проводилося для двох різних тестових версій коду програм, з та без використанням додаткових арифметичних функцій.

Інформація, щодо порівняльного співвідношення одиниць строчок програмного коду (без урахування не несучих інформацію рядків) наведена у табл. 4.1 та на рис. 4.1.

Для вимірювання часу виконання робочого тіла програми використовувався інструмент clock [40]. Усереднені результати за 100 запусків компілятора наведені у табл. 4.2 та на рис. 4.2.

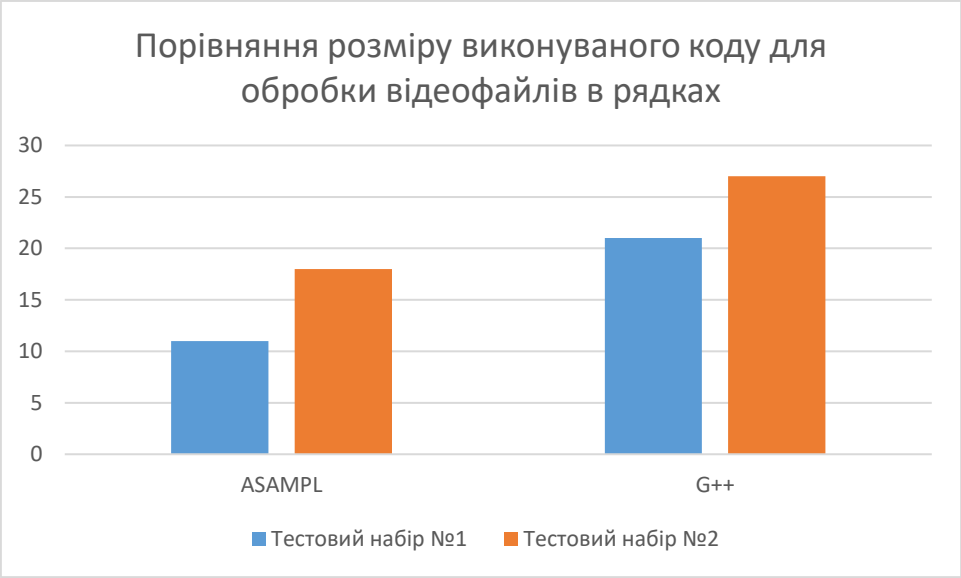


Рис. 4.1. Порівняння розміру виконуваного коду

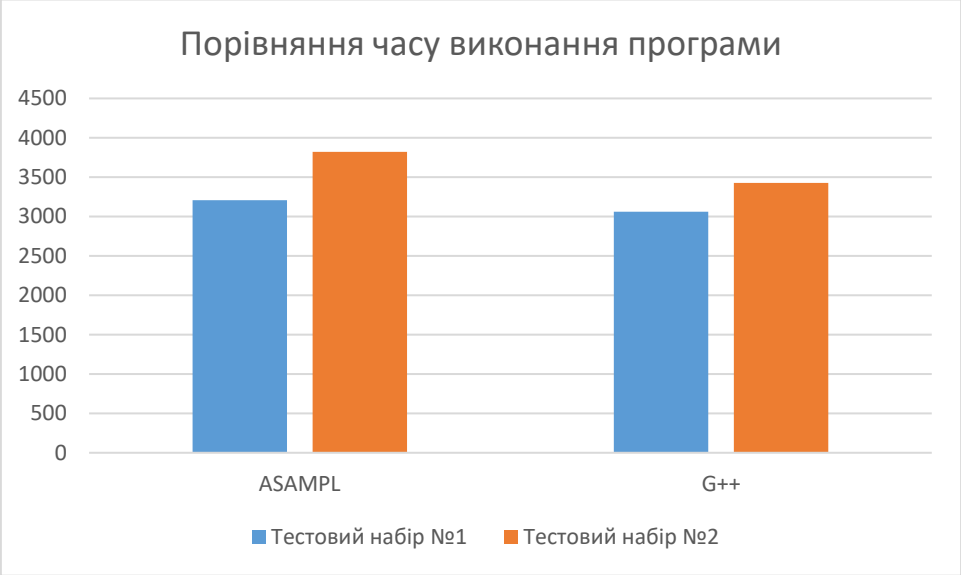


Рис. 4.2. Порівняння часу виконання

4.4. Напрямки подальшого вдосконалення

Мови програмування ASAMPL ще є куди вдосконалювати, і одним основних напрямків на думку автора проекту є покращення синтаксису даної мови програмування.

В першу чергу має сенс прибрати громіздкі конструкції призначені для визначення змінних, більшість розділів програми просто копіюють одне одного та забивають пам’ять перевірками правильності вхідного синтаксису, який є не таким вже і важливим.

Основним аргументом у підтримку цієї пропозиції є швидкий огляд самого опису даної граматики у форматі ebnf. В ньому можна побачити, що усі розділи підпорядковуються одному й тому ж принципу, який можна сформулювати наступним чином.

Ключове слово розділу, відкриття операторних дужок, будь яка кількість оголошень змінних, закриття операторних дужок.

На перший погляд може здатися, що такий спосіб оголошення змінних може полегшити роботу з ними, шляхом кращого структурування коду, але це не так. Наприклад, якщо змінних немає, відповідний розділ все одно потрібно виокремлювати, так як він є частиною базового синтаксису програми. Також багато розділів які не просто схожі, а є майже однаковими навіть з точки зору синтаксису оголошення змінної.

Тому моєю рекомендацією буде ввести якусь іншу систему оголошення змінних та підключення бібліотек.

Наприклад на мою думку було б доцільно виділяти об’явлення змінних, підключення бібліотек і так далі, спеціально зарезервованими для цього ключовими словами, наприклад let.

Такий запис безумовно б зробив обробку такого синтаксису, як візуальну, так і програмну набагато легшою а отже привабливішою для реалізації. А також зробив вигляд програми загалом менше схожим на застарілі мови програмування.

Друга моя пропозиція буде стосуватися форматів зберігання даних, серед яких має сенс особливо розглянути формат зберігання дати та часу, так як одним з основних напрямків даної мови є робота з даними у форматі реального часу.

Представлений формат запису даних не є прийнятним по двом причинам:

1. Розділення дати і часу на різні типи не є доцільним.
2. Формат запису цих даних є конфліктуючим з записом найбільш простих виразів по типу ділення.

Отже, по вищезазначеним пунктам, можна зробити висновок, що потрібно змінити формат зберігання дати та часу на більше простий та універсальний у використанні.

В даному випадку моєї пропозицією буде створення спеціального контейнеру, який буде містити усі необхідні поля для зберігання дати та часу у єдиному форматі, та містити набір базових функцій взаємодії з цими даними, який по мірі розвитку мови програмування можна розширювати та доповнювати.

ВИСНОВКИ

Головною метою обраного дипломного проекту було розроблення на основі наданого формального опису граматики мови обробки мультимедійних даних ASAMPL.

У першому розділі даної дипломної роботи було проведено аналіз існуючих технологій, призначених для оброблення різних мультимедійних даних.

На основі цього дослідження було зроблено висновок, що рівень розвитку апаратної частини технологій, що призначені для зчитування та обробки мультимедійних даних є достатньо високим, однак у той же самий час рівень розвитку програмних засобів, що ефективно реалізують можливості апаратних платформ є значно нижчим та набагато менше доступним для пересічного користувача подібних систем.

Більш технологічно складні рішення задач вірогідно також існують, але доступ до подібних рішень як правило закритий для пересічного користувача подібних програм.

Отже, проведені дослідження підтвердили теорію про необхідність розробити єдине програмне рішення, для уніфікованої обробки різноманітних мультимедійних даних.

Наступним пунктом було вирішено розглянути засоби реалізації, що могли б підходити для більш ефективної реалізації обраного програмного продукту, через призму сформулюваних на початку відповідного розділу.

На основі розглянутих даних було прийнято рішення використовувати статично типізовану мову програмування C++, на якому вирішено написати модулі парсингу вхідного потоку символів, впорядковане приведення їх до вигляду абстрактного синтаксичного дерева, та подальше виконання розібраних команд.

Наступним кроком було розроблення та написання самого програмного продукту, а також зіставлення об’ємного описання методу

роботи кожного з окремих модулів для полегшення подальшого розширення та доповнення даного програмного продукту.

У кінцевому розділі було проведено аналіз розробленого програмного продукту, за критеріями часу виконання робочого коду, також за критеріями розміру спеціалізованого синтаксису, призначеного для оброблення мультимедійних даних.

У фінальній частині написані можливі шляхи подальшого розвитку програмного забезпечення, та проведення оптимізації окремих модулів програмного продукту.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. The State of 3D Printing [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://www.forbes.com/sites/louiscolumbus/2018/05/30/the-state-of-3d-printing-2018/#668f6bf67b0a – Дата доступу: 20.05.2019 – Назва з екрану.
2. Media Foundation [Електронний ресурс]. – 2019 . – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Media_Foundation – Дата доступу: 20.05.2019 – Назва з екрану.
3. DirectShow [Електронний ресурс]. – 2018 . – Режим доступу до ресурсу: https://docs.microsoft.com/enus/windows/desktop/directshow/ – Дата доступу: 20.05.2019 – Назва з екрану.
4. Intel Media Software Development Kit [Електронний ресурс]. – 2019 . – Режим доступу до ресурсу: https://itpro.ua/product/intel-media-software-development-kit-intel-media-sdk/?tab=description – Дата доступу: 20.05.2019 – Назва з екрану.
5. Media Foundation Architecture [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/windows/desktop/medfound/media-foundation-architecture – Дата доступу: 20.05.2019 – Назва з екрану.
6. DRM Security & Information Access Control for Document Protection [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://www.locklizard.com/drm-security/ – Дата доступу: 20.05.2019 – Назва з екрану.
7. DirectX Video Acceleration [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: http://bit.do/DirectX_Video_Acceleration – Дата доступу: 20.05.2019 – Назва з екрану.
8. Multimedia Class Scheduler Service [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://docs.microsoft.com/en-

us/windows/desktop/procthread/multimedia-class-scheduler-service – Дата доступу: 20.05.2019 – Назва з екрану.

9. FFmpeg documentation [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://ffmpeg.org/documentation.html – Дата доступу: 20.05.2019 – Назва з екрану.
10. FFmpeg supported platforms [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/FFmpeg – Дата доступу: 20.05.2019 – Назва з екрану.
11. Media Lovin' Toolkit [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://www.mltframework.org/ – Дата доступу: 20.05.2019 – Назва з екрану.
12. Commission, E. Technology readiness levels. Horizon 2020, Work Programme 2014–2015, GeneralAnnexes, Extract from Part 19, Commission Decision C(2014)4995 [Електронний ресурс]. – 2014 . – Режим доступу до ресурсу: http://ec.europa.eu/research/participants/data/ref/h2020/wp/2014_2015/annexes/h2020-wp1415-annex-g-trl_en.pdf – Дата доступу: 20.05.2019 – Назва з екрану.
13. TIOBE Index for June 2019 [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://www.tiobe.com/tiobe-index/ – Дата доступу: 20.05.2019 – Назва з екрану.
14. Most Popular Programming Languages [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://stackify.com/popular-programming-languages-2018/ – Дата доступу: 20.05.2019 – Назва з екрану.
15. Java Virtual Machine Security [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://help.sap.com/doc/saphelp_snc_uiaddon_10/1.0/en-US/55/c602ccca7441afa6d00e088e4011de/content.htm?no_cache=true – Дата доступу: 20.05.2019 – Назва з екрану.
16. Comparison of Java and C++ [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: http://bit.do/Comparison_of_Java_and_C – Дата доступу: 20.05.2019 – Назва з екрану.

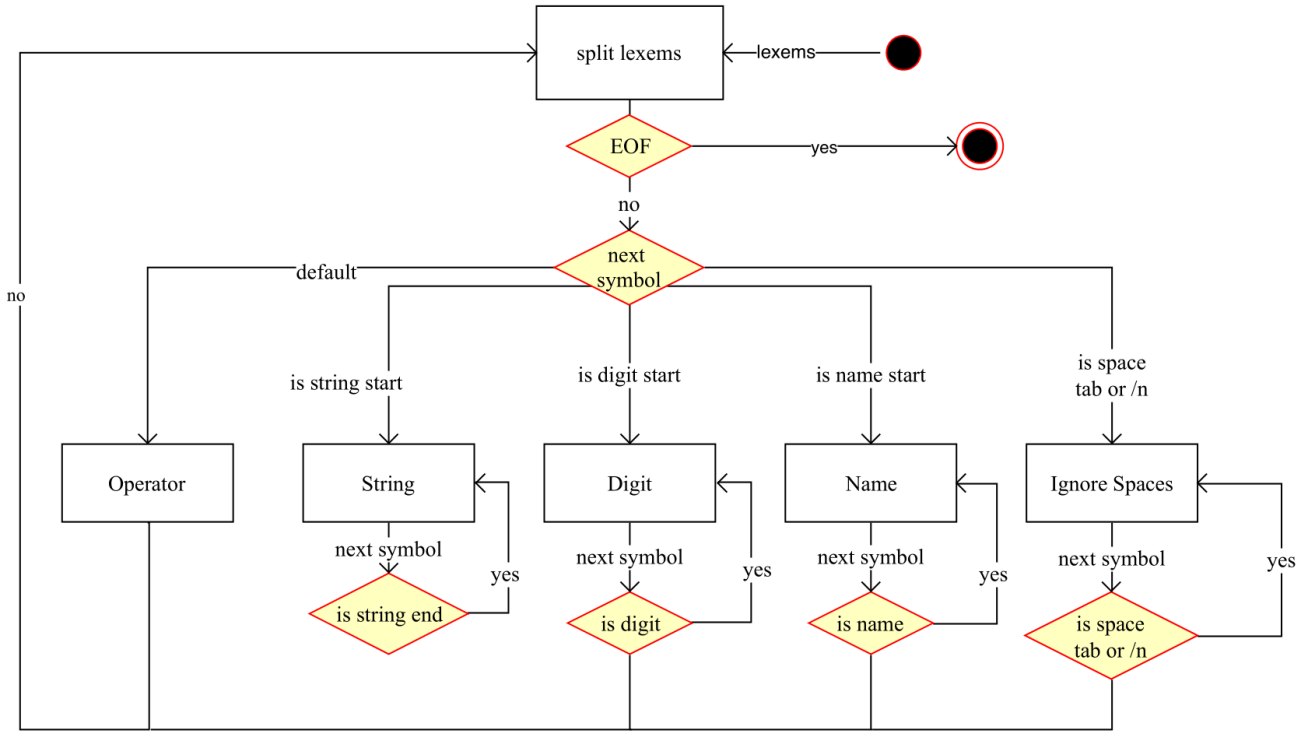
17. В Google провели порівняння продуктивності C ++, Java, Go і Scala [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: http://www.opennet.ru/opennews/art.shtml?num=30784 – Дата доступу: 20.05.2019 – Назва з екрану.
18. What is Comand Line Interface? [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://www.w3schools.com/whatis/whatis_cli.asp – Дата доступу: 20.05.2019 – Назва з екрану.
19. What Is the Microsoft .NET Framework [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://www.howtogeek.com/253588/what-is-the-microsoft-net-framework-and-why-is-it-installed-on-my-pc/ – Дата доступу: 20.05.2019 – Назва з екрану.
20. Introduction to Unix [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://www.softwaretestinghelp.com/unix-introduction/ – Дата доступу: 20.05.2019 – Назва з екрану.
21. Features of C language [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://www.studytonight.com/c/features-of-c.php – Дата доступу: 20.05.2019 – Назва з екрану.
22. GCC [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://gcc.gnu.org/ – Дата доступу: 20.05.2019 – Назва з екрану.
23. GLib Overview [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://developer.gnome.org/glib/stable/ – Дата доступу: 20.05.2019 – Назва з екрану.
24. Gtk Overview [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://www.gtk.org/ – Дата доступу: 20.05.2019 – Назва з екрану.
25. What are the advantages of C++ Programming Language? [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://www.tutorialspoint.com/What-are-the-advantages-of-Cplusplus-Programming-Language – Дата доступу: 20.05.2019 – Назва з екрану.

26. Templates [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: http://www.cplusplus.com/doc/oldtutorial/templates/ – Дата доступу: 20.05.2019 – Назва з екрану.
27. Common Language Runtime (CLR) overview [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/dotnet/standard/clr – Дата доступу: 20.05.2019 – Назва з екрану.
28. Base Class Library [Електронний ресурс]. – 2019 . – Режим доступу до ресурсу: https://www.nuget.org/packages/Microsoft.Bcl/ – Дата доступу: 20.05.2019 – Назва з екрану.
29. C Sharp 3.0 [Електронний ресурс]. – 2019 . – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/C_Sharp_3.0 – Дата доступу: 20.05.2019 – Назва з екрану.
30. Benchmark [Електронний ресурс]. – 2019 . – Режим доступу до ресурсу: https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/csharpcore-gpp.html – Дата доступу: 20.05.2019 – Назва з екрану.
31. Python [Електронний ресурс]. – 2019 . – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Python_(programming_language)
32. Cpython [Електронний ресурс]. – 2019 . – Режим доступу до ресурсу: https://github.com/python/cpython – Дата доступу: 20.05.2019 – Назва з екрану.
33. Distributed Component Object Model [Електронний ресурс]. – 2019 . – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-dcom/4a893f3d-bd29-48cd-9f43-d9777a4415b0 – Дата доступу: 20.05.2019 – Назва з екрану.
34. Jython [Електронний ресурс]. – 2019 . – Режим доступу до ресурсу: https://www.jython.org/ – Дата доступу: 20.05.2019 – Назва з екрану.
35. Ironpython [Електронний ресурс]. – 2019 . – Режим доступу до ресурсу: https://ironpython.net/ – Дата доступу: 20.05.2019 – Назва з екрану.

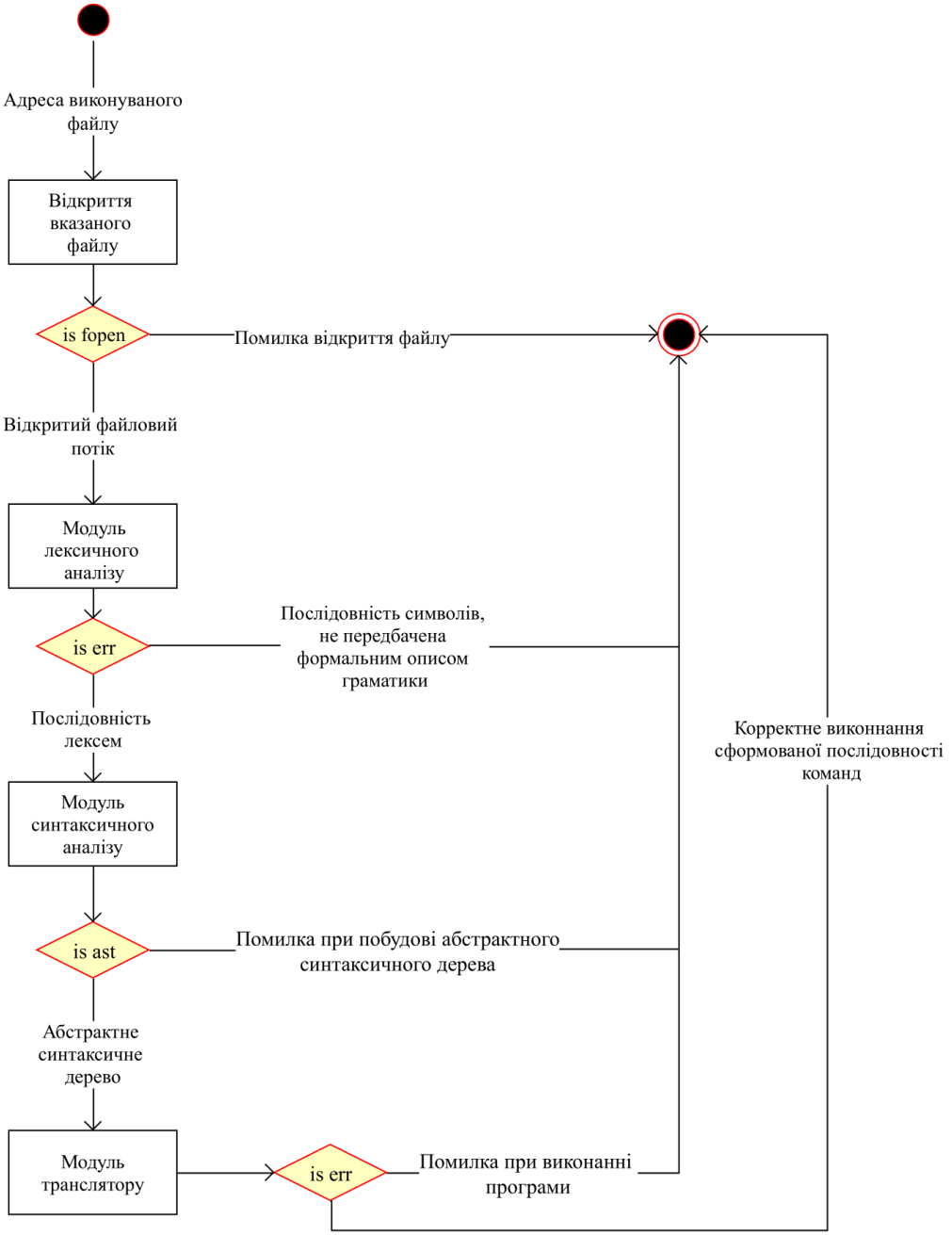
36. Pythonnet [Електронний ресурс]. – 2019 . – Режим доступу до ресурсу: http://pythonnet.github.io/ – Дата доступу: 20.05.2019 – Назва з екрану.
37. Recursive descent parser [Електронний ресурс]. – 2019 . – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Recursive_descent_parser – Дата доступу: 20.05.2019 – Назва з екрану.
38. Left recursion [Електронний ресурс]. – 2019 . – Режим доступу до ресурсу: https://www.gatevidyalay.com/left-recursion-left-recursion-elimination/ – Дата доступу: 20.05.2019 – Назва з екрану.
39. OpenCv [Електронний ресурс]. – 2019 . – Режим доступу до ресурсу: https://opencv.org/ – Дата доступу: 20.05.2019 – Назва з екрану.
40. Clock program [Електронний ресурс]. – 2019 . – Режим доступу до ресурсу: http://www.cplusplus.com/reference/ctime/clock/ – Дата доступу: 20.05.2019 – Назва з екрану.

ДОДАТКИ

Додаток 1
Копії графічних матеріалів

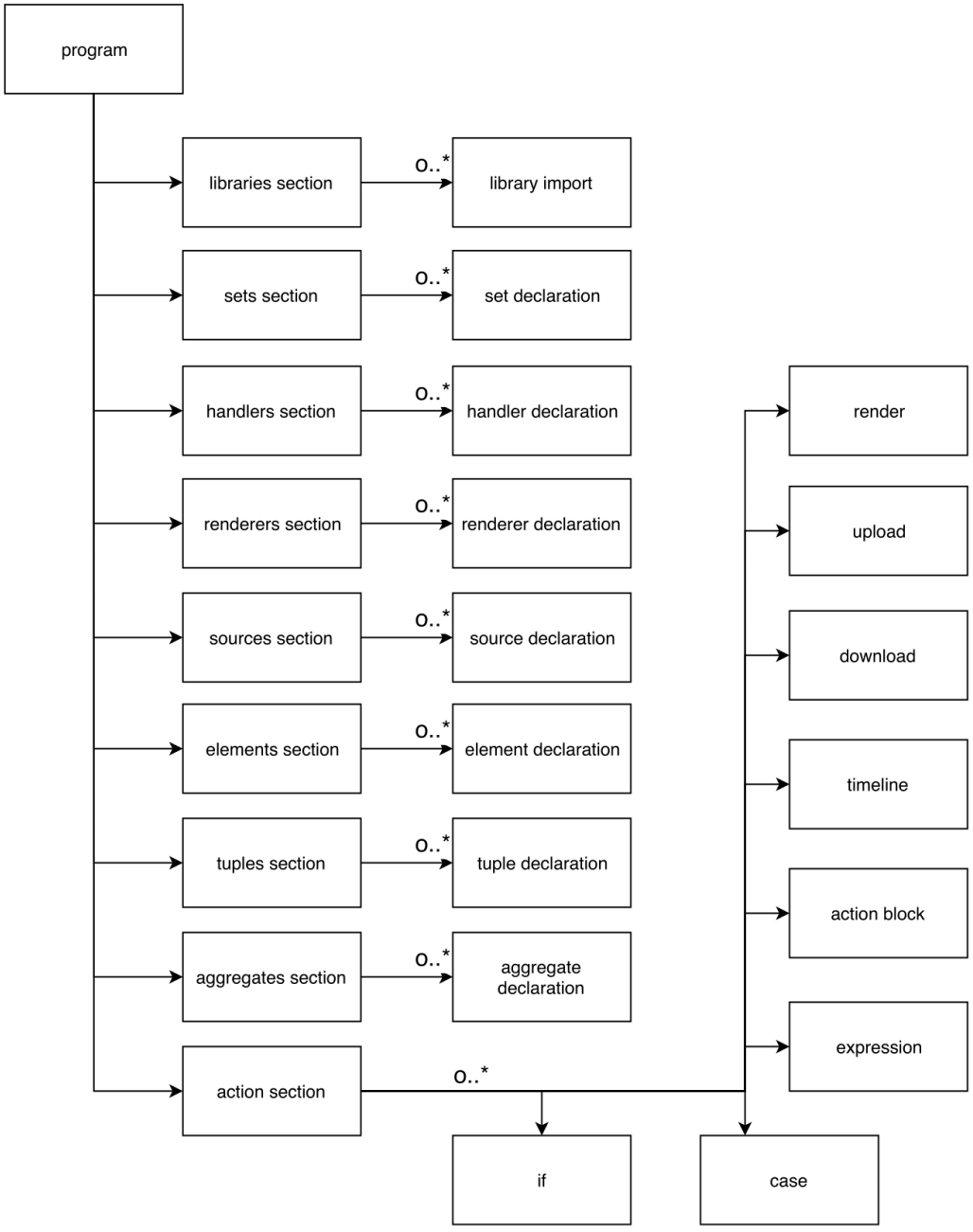


ДП.045480-06-99
Компілятор мови ASAMPL.
Діаграма діяльності модулю лексичного аналізу.
UML діаграма діяльності



ДП.045480-07-99
Компілятор мови ASAMPL.
Діаграма діяльності компілятора мови ASAMPL. UML діаграма діяльності

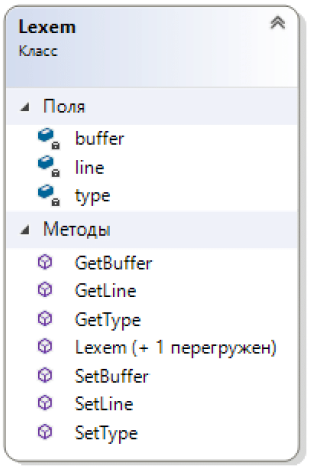
Узагальнена схема розташування листків абстрактного синтаксичного дерева



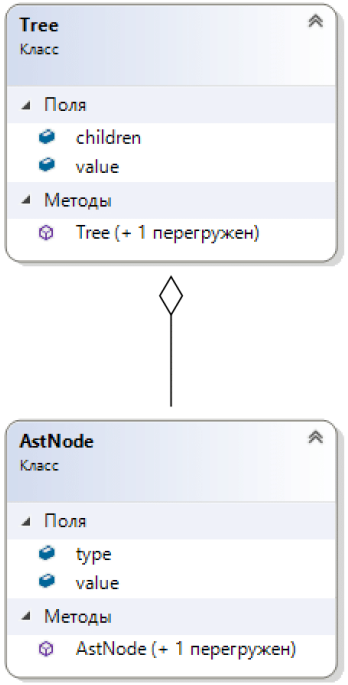
Песчанський В.Ю., група КП-52

Діаграми класів основних компонентів розробленого ПЗ

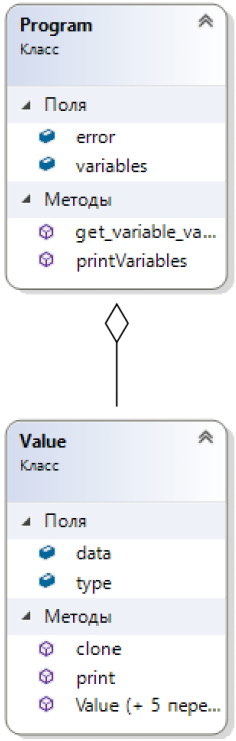
Діаграма класів лексеру



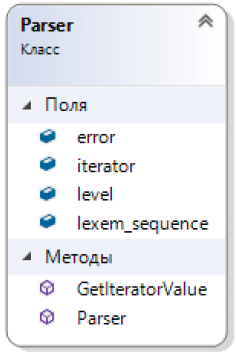
Діаграма класів AST



Діаграма класів інтерпритатору



Діаграма класів парсеру



Песчанський В.Ю., група КП-52

Додаток 2
Лістинги програми

Модуль лексичного аналізу. Лістинг циклу оброблення вхідного потоку символів.

```
int Lexer_splitTokens(std::fstream *fs, std::vector<Lexem>* lexem_sequence) {

	if (fs == NULL) return 1;

	while (fs->peek() != EOF) {

		Lexem *lexem_to_add = new Lexem();

		IgnoreTabNewlinesAndSpaces(fs);

		if (IS_NAME_START(fs->peek()))
		{
			LexemName(fs, lexem_to_add);
		}

		else if (isdigit(fs->peek()))
		{
			LexemDigit(fs, lexem_to_add);
		}

		else if (IS_STRING_LITERAL_START(fs->peek()))
		{
			LexemStringLiteral(fs, lexem_to_add);
		}

		else {
			LexemOperator(fs, lexem_to_add);
		}

		lexem_sequence->push_back(*lexem_to_add);
	}

	return 0;
}
```

Модуль синтаксичного аналізу. Лістинг функції перетворення вхідної послідовності лексем у AST.

```
Tree* parser_buid_tree(std::vector<Lexem>* lexem_sequence) {

	Parser parser;
	parser.lexem_sequence = lexem_sequence;

	parser.iterator = lexem_sequence->begin();

	parser.level = -1;

	Tree * tree = program(&parser);

	if (!parser.error.empty()) {

		std::cout << parser.error << std::endl;
		return NULL;

	}
	return tree;

}
```

Модуль трансляції. Лістинг модулю виконання послідовності команд, представленої абстрактним синтаксичним деревом.

```
static Value * execute_expression(Program * program, Tree * node);

void execute_action(Program * program, Tree * childNode);
void execute_block(Program * program, Tree * blockTreeNode);
void execute_if(Program * program, Tree * node);

void execute_timeline(Program * program, Tree * node);
void execute_substitution(Program * program, Tree * node);
void execute_sequence(Program * program, Tree * node);
void execute_download(Program * program, Tree * node);
void execute_upload(Program * program, Tree * node);
void execute_render(Program * program, Tree * node);



void execute_library_import(Program * program, Tree * node);

void execute_handler_import(Program * program, Tree * node);
void execute_renderer_declaration(Program * program, Tree * node);
void execute_source_declaration(Program * program, Tree * node);
void execute_aggregate_declaration(Program * program, Tree * node);
void execute_set_declaration(Program * program, Tree * node);
void execute_element_declaration(Program * program, Tree * node);
void execute_tuple_declaration(Program * program, Tree * node);
void execute_aggregate_declaration(Program * program, Tree * node);

void execute_action(Program * program, Tree * node);

int tm_to_sec(std::tm tm) {
   return tm.tm_sec + tm.tm_min * 60 + tm.tm_hour * 360;
}


double is_number(Value * self) {
   assert(self->type == ValueType_NUMBER);
   return *((double*)self->data);
}

bool is_bool(Value * self) {
   assert(self->type == ValueType_BOOL);
   return *((bool*)self->data);
}

char * is_string(Value * self) {
   assert(self->type == ValueType_STRING);
   return ((char*)self->data);
}

typedef bool(*comparison_method)(Value * a, Value * b);

bool more_or_less(Value * a, Value * b) {

   if (a->type != b->type) assert(0 && "Not supposed");;
   switch (a->type) {
         case ValueType_NUMBER: return is_number(a) < is_number(b);
         default: assert(0 && "Not supposed");
   }
}
```

```
   bool more_or_less_equal(Value * a, Value * b) {
      if (a->type != b->type) assert(0 && "Not supposed");;
      switch (a->type) {
            case ValueType_NUMBER: return is_number(a) <= is_number(b);
            case ValueType_BOOL: return is_bool(a) == is_bool(b);

            default: assert(0 && "Not supposed");
      }
   }

   bool equal(Value * a, Value * b) {
      if (a->type != b->type) return false;
      switch (a->type) {
            case ValueType_BOOL: return is_bool(a) == is_bool(a);
            case ValueType_NUMBER: return fabs(is_number(a) - is_number(b))
< 1e-6;
            case ValueType_STRING: return is_string(a) == is_string(b);

            case ValueType_UNDEFIND: return true;
            default: assert(0 && "Not supposed");
      }
   }

   bool value_to_bool(Value * self) {
      switch (self->type) {
      case ValueType_BOOL: return is_bool(self);
      case ValueType_NUMBER: return fabs(is_number(self)) > 1e-6;
      case ValueType_STRING: return is_string(self) != NULL;
      case ValueType_UNDEFIND: return false;
      default: assert(0 && "Not implemented");
      }
   }

   bool compare(Program * program, Tree * node, comparison_method funk) {
      Tree * firstChild = node->children[0];
      Value * firstValue = execute_expression(program, firstChild);
      if (!program->error.empty()) return NULL;

      Tree * secondChild = node->children[1];
      Value * secondValue = execute_expression(program, secondChild);
      if (!program->error.empty()) return NULL;

      bool res = funk(firstValue, secondValue);
      return res;
   }


   static Value * execute_expression(Program * program, Tree * node) {

      AstNode * astNode = node->value;

      switch (astNode->type)
      {

      case AstNodeType_NUMBER: {
            double number = atof(astNode->value.c_str());
            //std::cout << number << std::endl;
            return new Value(number);
      }

      case AstNodeType_STRING: {
            return new Value(astNode->value);
      }
```

```
      case AstNodeType_BOOL: {
            return new Value(astNode->value == "true" ? true : false);
      }

      case AstNodeType_ADD: {
            if (!node->children.empty()) {
                  Tree * firstChild = node->children[0];
                  Value    *    firstValue    =    execute_expression(program,
firstChild);

                  if (!program->error.empty()) return NULL;
                  if (firstValue->type != ValueType_NUMBER) {
                        program->error = "Invalid operation";
                        return NULL;
                  }

                  if (node->children.size() == 1) {
                        return firstValue;
                  }

                  else {
                        Tree * secondChild = node->children[1];
                        Value  *  secondValue  =  execute_expression(program,
secondChild);

                        if (!program->error.empty()) return NULL;
                        if (secondValue->type != ValueType_NUMBER) {
                              program->error = "Invalid operation";
                              return NULL;
                        }
                        double     res     =     is_number(firstValue)     +
is_number(secondValue);

                        return new Value(res);
                  }
            }
            return new Value();
      }

      case AstNodeType_MUL: {
            if (!node->children.empty()) {
                  Tree * firstChild = node->children[0];
                  Value    *    firstValue    =    execute_expression(program,
firstChild);
                  if (!program->error.empty()) return NULL;
                  if (firstValue->type != ValueType_NUMBER) {
                        program->error = "Invalid operation";
                        return NULL;
                  }
                  Tree * secondChild = node->children[1];
                  Value    *    secondValue    =    execute_expression(program,
secondChild);
                  if (!program->error.empty()) return NULL;
                  if (secondValue->type != ValueType_NUMBER) {
                        program->error = "Invalid operation";
                        return NULL;
                  }
                  double       res       =       is_number(firstValue)       *
is_number(secondValue);
                  return new Value(res);
            }
            return new Value();
      }
```

```
      case AstNodeType_DIV: {
            if (!node->children.empty()) {
                  Tree * firstChild = node->children[0];
                  Value    *    firstValue    =    execute_expression(program,
firstChild);

                  if (!program->error.empty()) return NULL;
                  if (firstValue->type != ValueType_NUMBER) {
                        program->error = "Invalid operation";
                        return NULL;
                  }
                  Tree * secondChild = node->children[1];
                  Value    *    secondValue    =    execute_expression(program,
secondChild);
                  if (!program->error.empty()) return NULL;
                  if (secondValue->type != ValueType_NUMBER) {
                        program->error = "Invalid operation";
                        return NULL;
                  }
                  if (fabs(is_number(secondValue)) < 1e-6) {
                        program->error = "Invalid operation";
                        return NULL;
                  }

                  double       res       =       is_number(firstValue)       /
is_number(secondValue);

                  return new Value(res);
            }
            return new Value();
      }

      case AstNodeType_SUB: {
            if (!node->children.empty()) {
                  Tree * firstChild = node->children[0];
                  Value    *    firstValue    =    execute_expression(program,
firstChild);
                  if (!program->error.empty()) return NULL;
                  if (firstValue->type != ValueType_NUMBER) {
                        program->error = "Invalid operation";
                        return NULL;
                  }

                  if (node->children.size() == 1) {
                        double res = -is_number(firstValue);
                        return new Value(res);;
                  }
                  else {
                        Tree * secondChild = node->children[1];
                        Value  *  secondValue  =  execute_expression(program,
secondChild);
                        if (!program->error.empty()) return NULL;
                        if (secondValue->type != ValueType_NUMBER) {
                              program->error = "Invalid operation";
                              return NULL;
                        }

                        double     res     =     is_number(firstValue)     -
is_number(secondValue);
                        return new Value(res);
                  }
            }
            return new Value();
      }
```

```
	case AstNodeType_AND: {
		Tree * firstChild = node->children[0];
		Value * firstValue = execute_expression(program, firstChild);
		if (!program->error.empty()) return NULL;

		Tree * secondChild = node->children[1];
		Value * secondValue = execute_expression(program, secondChild);
		if (!program->error.empty()) return NULL;

		bool	res	=	value_to_bool(firstValue)	&&
value_to_bool(secondValue);

		return new Value(res);
	}
	case AstNodeType_OR: {
		Tree * firstChild = node->children[0];
		Value * firstValue = execute_expression(program, firstChild);
		if (!program->error.empty()) return NULL;
		Tree * secondChild = node->children[1];
		Value * secondValue = execute_expression(program, secondChild);
		if (!program->error.empty()) return NULL;

		bool	res	=	value_to_bool(firstValue)	||
value_to_bool(secondValue);

		return new Value(res);
	}
	case AstNodeType_EQUAL: {
		bool eq = compare(program, node, equal);
		return  new Value(eq);
	}
	case AstNodeType_NOTEQUAL: {
		bool eq = compare(program, node, equal);
		return  new Value(!eq);
	}
	case AstNodeType_LESS: {
		bool eq = compare(program, node, more_or_less);
		return  new Value(eq);
	}
	case AstNodeType_MORE: {
		bool eq = compare(program, node, more_or_less);
		return  new Value(!eq);
	}
	case AstNodeType_LESS_OR_EQUAL: {
		bool eq = compare(program, node, more_or_less_equal);
		return  new Value(eq);
	}
	case AstNodeType_MORE_OR_EQUAL: {
		bool eq = compare(program, node, more_or_less_equal);
		return  new Value(!eq);
	}
	case AstNodeType_ASSIGN: {
		Tree * firstChildNode = node->children[0];
		AstNode * firstChild = firstChildNode->value;

		if (firstChild->type != AstNodeType_ID) {
			program->error = "Can't assign to rvalue";
			return NULL;
		}
		std::string varId = firstChild->value;
		//
		Tree * secondChild = node->children[1];
		Value * secondValue = execute_expression(program, secondChild);
```

```
            if (!program->error.empty()) return NULL;
            //
            if (program->variables.find(varId) == program->variables.end())
{
                  program->error = "Var for assign not found";
                  delete(secondValue);
                  return NULL;
            }

            //Value * oldValue = Dict_set(program->variables, varId,
Value_newCopy(secondValue));
            Value * oldValue = program->variables[varId];
            program->variables[varId] = secondValue->clone();

            delete(oldValue);
            return secondValue;
            break;
      }

      case AstNodeType_ID: {

            std::string varId = astNode->value;
      /*    if (!node->children.empty()) {
                  Tree * argListNode = node->children[0];
                  AstNode * argList = argListNode->value;
                  assert(argList->type == AstNodeType_ARGLIST);
                  //
                  if     (program->variables.find(varId)     ==     program-
>variables.end()) {
                        program->error = strdup("Call unknown function");
                        return NULL;
                  }
                  //
                  List * arguments = List_new();
                  for (int i = 0; i < List_count(argListNode->children); i++)
{
                        Tree   *   argListChildNode   =   List_at(argListNode-
>children, i);
                        Value     *     argumentValue     =     eval(program,
argListChildNode);

                        if (program->error) {
                              // @todo free all arguments values
                              List_free(arguments);
                              return NULL;
                        }
                        List_add(arguments, argumentValue);
                  }
                  // call function

                  StdFunction  *  func  =  Dict_get(program->functions,  (char
*)varId);
                  Value * retValue = func->ptr(program, arguments);

                  for (int i = 0; i < List_count(arguments); i++) {
                        Value * argListChildNode = List_at(arguments, i);
                        if (argListChildNode->type != ValueType_ARRAY) {
                              Value_free(argListChildNode);
                        }
                  }

                  List_free(arguments);
                  return retValue;
            }*/
```

```
            Value * varValue = program->get_variable_value_by_id(varId);
            if (!program->error.empty()) return NULL;
            return varValue->clone();
      }

      case AstNodeType_ARGLIST: {

      }

      default:
            assert("Not implemented");

            break;
      }
      return NULL;
   }

   void execute_if(Program * program, Tree * node) {

      Tree * exprNode = node->children[0];

      Value * testValue = execute_expression(program, exprNode);

      if (!program->error.empty()) return;

      bool _bool = value_to_bool(testValue);

      delete(testValue);
      if (_bool) {
            execute_block(program, node->children[1]);
      }
      else if (node->children.size() > 2) {
            execute_block(program, node->children[2]);
      }
   }

   void execute_timeline(Program * program, Tree * node)
   {
      Tree * conditionNode = node->children[0];

      Tree * actionsNode = node->children[1];



      switch (conditionNode->value->type) {

      case AstNodeType_TIMELINE_EXPR:

            execute_block(program, actionsNode);
            break;

      case AstNodeType_TIMELINE_AS: {

            Value * value = execute_expression(program, conditionNode-
>children[0]);

            if (!program->error.empty()) return;

            if (value->type != ValueType_STRING) {
                  program->error = "Invalid timeline argument";
                  return;
            }

            std::string value_d = *(std::string *)value->data;
```

```
            delete(value);

            struct std::tm timer = { 0 };
            std::istringstream ss(value_d);

            ss >> std::get_time(&timer, "%H:%M:%S");

            std::time_t begin;
            std::time_t now;

            time(&begin);

            while (true) {
                  time(&now);
                  if ( difftime(now, begin) > tm_to_sec(timer)) break;

                  execute_block(program, actionsNode);
            }

      }
      break;

      case AstNodeType_TIMELINE_UNTIL:

            while (true) {
                  Value    *    testValue    =    execute_expression(program,
conditionNode->children[0]);
                  bool testBool = value_to_bool(testValue);
                  delete(testValue);

                  if (!testBool) break;

                  execute_block(program, actionsNode);
                  if (!program->error.empty()) return;
            }
      }

   }

   void execute_substitution(Program * program, Tree * node)
   {
   }

   void execute_sequence(Program * program, Tree * node)
   {
   }

   void execute_download(Program * program, Tree * node)
   {

      Tree * variableNode = node->children[0];
      AstNode * _variableNode = variableNode->value;

      Tree * sourceNode = node->children[1];
      Value * value = execute_expression(program, sourceNode);
      if (!program->error.empty()) return;
      if (value->type != ValueType_STRING) {
            program->error = "Invalid download argument";
            return;
      }
      std::string source = *(std::string *)value->data;
      delete(value);
```

```
    std::string varId = _variableNode->value;

    cv::VideoCapture cap(source);
    //cv::VideoCapture cap("E:/video.mp4");

    if (!cap.isOpened()) {
          program->error = "Error opening stream or file";
          return;
    }
    /*while (1) {
          cv::Mat frame;
          cap >> frame;

          if (frame.empty()) break;

          cv::imshow("Frame", frame);
          char c = (char)cv::waitKey(25);
          if (c == 27)
                break;

    }*/


    Value * varValue = new Value(cap);

    //cv::VideoCapture cap2 = *(cv::VideoCapture *)varValue->data;

    if (program->variables.find(varId) != program->variables.end()) {
          program->error = "Duplicate variable id";
          delete(varValue);
          return;
    }
    program->variables.emplace(varId, varValue);
    //program->printVariables();

    //Tree * handlerNode = node->children[2];



}

void execute_upload(Program * program, Tree * node)
{
    Value * sourceValue = execute_expression(program, node->children[0]);
    if (!program->error.empty()) return;
    if (sourceValue->type != ValueType_VIDEO) return;

    cv::VideoCapture cap = *(cv::VideoCapture *)sourceValue->data;


    Tree * pathNode = node->children[1];
    Value * _pathNode = execute_expression(program, pathNode);

    if (!program->error.empty()) return;
    if (_pathNode->type != ValueType_STRING) {
          program->error = "Invalid download argument";
          return;
    }
    std::string path = *(std::string *)_pathNode->data;
    delete(_pathNode);

    int frame_width = cap.get(cv::CAP_PROP_FRAME_WIDTH);
    int frame_height = cap.get(cv::CAP_PROP_FRAME_HEIGHT);
```

```
      cv::VideoWriter video(path, cv::VideoWriter::fourcc('M', 'J', 'P',
'G'), 30, cv::Size(frame_width, frame_height));
      while (1)
      {
            cv::Mat frame;
            cap >> frame;

            if (frame.empty())
                  break;
            video.write(frame);

            char c = (char)cv::waitKey(1);
            if (c == 27)
                  break;
      }
      video.release();
   }

   void execute_render(Program * program, Tree * node)
   {
      Value * sourceValue = execute_expression(program, node->children[0]);
      if (!program->error.empty()) return;

      //Value   *   handlerValue   =   execute_expression(program,   node-
>children[1]);
      //if (!program->error.empty()) return;

      if (sourceValue->type != ValueType_VIDEO) return;

      cv::Mat frame;

      *(cv::VideoCapture *)sourceValue->data >> frame;

      if (frame.empty()) return;

      imshow("Frame", frame);

      char c = (char)cv::waitKey(25);
      if (c == 27)
            return;
   }



   void execute_block(Program * program, Tree * blockTreeNode) {
      for (int i = 0; i < blockTreeNode->children.size(); i++) {
            Tree * childNode = blockTreeNode->children[i];

            AstNode * child = childNode->value;
            execute_action(program, childNode);
            if (!program->error.empty()) break;
      }
   }

   void execute_library_import(Program * program, Tree * node)
   {
   }

   void execute_handler_import(Program * program, Tree * node)
   {
   }
```

```
   void execute_renderer_declaration(Program * program, Tree * node)
   {
   }

   void execute_source_declaration(Program * program, Tree * node)
   {
      for (auto &child : node->children) {


            Tree * fChildNode = child->children[0];
            AstNode * fChild = fChildNode->value;

            Tree * sChildNode = child->children[1];
            AstNode * sChild = sChildNode->value;
            //
            std::string varId = fChild->value;

            Value * varValue = execute_expression(program, sChildNode);

            if (!program->error.empty()) break;
            //
            if (program->variables.find(varId) != program->variables.end())
{
                  program->error = "Duplicate variable id";
                  delete(varValue);
                  break;
            }

            program->variables.emplace(varId, varValue);
      }
   }


   void execute_set_declaration(Program * program, Tree * node)
   {
   }

   void execute_element_declaration(Program * program, Tree * node)
   {


      for (auto &child : node->children) {


            Tree * fChildNode = child->children[0];
            AstNode * fChild = fChildNode->value;

            Tree * sChildNode = child->children[1];
            AstNode * sChild = sChildNode->value;
            //
            std::string varId = fChild->value;

            Value * varValue = execute_expression(program, sChildNode);

            if (!program->error.empty()) break;
            //
            if (program->variables.find(varId) != program->variables.end())
{
                  program->error = "Duplicate variable id";
                  delete(varValue);
                  break;
            }

            program->variables.emplace(varId, varValue);
```

```
   }

}


void execute_tuple_declaration(Program * program, Tree * node)
{
}

void execute_aggregate_declaration(Program * program, Tree * node)
{
}


void execute_action(Program * program, Tree * childNode) {
   AstNode * child = childNode->value;
   switch (child->type) {

   case AstNodeType_IF: {
         execute_if(program, childNode);
         if (!program->error.empty()) return;
         break;
   }

   case AstNodeType_SEQUENCE: {

         execute_sequence(program, childNode);
         if (!program->error.empty()) return;
         break;
   }

   case AstNodeType_DOWNLOAD: {
         execute_download(program, childNode);
         if (!program->error.empty()) return;
         break;
   }
   case AstNodeType_UPLOAD: {
         execute_upload(program, childNode);
         if (!program->error.empty()) return;
         break;
   }
   case AstNodeType_RENDER: {
         execute_render(program, childNode);
         if (!program->error.empty()) return;
         break;
   }
   case AstNodeType_TIMELINE: {
         execute_timeline(program, childNode);
         if (!program->error.empty()) return;
         break;
   }
   case AstNodeType_SUBSTITUTION: {
         execute_substitution(program, childNode);
         if (!program->error.empty()) return;
         break;
   }
   case AstNodeType_BLOCK: {
         execute_block(program, childNode);
         if (!program->error.empty()) return;
         break;
   }
   default: {
         Value * val = execute_expression(program, childNode);
         if (!program->error.empty()) return;
```

```
        delete(val);
    }
    }
}


int execute(Tree * astTree)
{

   AstNode * astNode = astTree->value;
   assert(astNode->type == AstNodeType_PROGRAM);
   Program program;

   for (auto &child : astTree->children) {
         AstNode * childNode = child->value;

         switch (childNode->type) {
         case AstNodeType_LIBRARIES: {
         //    std::cout << "AstNodeType_LIBRARIES"  << std::endl;
               execute_library_import(&program, child);
               if (!program.error.empty()) break;
               break;
         }
         case AstNodeType_HANDLERS: {
               //std::cout << "AstNodeType_HANDLERS" << std::endl;
               execute_handler_import(&program, child);
               if (!program.error.empty()) break;
               break;
         }
         case AstNodeType_RENDERERS: {
               //std::cout << "AstNodeType_RENDERERS" << std::endl;
               execute_renderer_declaration(&program, child);
               if (!program.error.empty()) break;
               break;
         }
         case AstNodeType_SOURCES: {
               //std::cout << "AstNodeType_SOURCES" << std::endl;
               execute_source_declaration(&program, child);
               if (!program.error.empty()) break;
               break;
         }
         case AstNodeType_SETS: {
               //std::cout << "AstNodeType_SETS" << std::endl;
               execute_set_declaration(&program, child);
               if (!program.error.empty()) break;
               break;
         }
         case AstNodeType_ELEMENTS: {
               //std::cout << "AstNodeType_ELEMENTS" << std::endl;
               execute_element_declaration(&program, child);
               if (!program.error.empty()) break;
               break;
         }
         case AstNodeType_TUPLES: {
               //std::cout << "AstNodeType_TUPLES" << std::endl;
               execute_tuple_declaration(&program, child);
               if (!program.error.empty()) break;
               break;
         }
         case AstNodeType_AGGREGATES: {
               //std::cout << "AstNodeType_AGGREGATES" << std::endl;
               execute_aggregate_declaration(&program, child);
               if (!program.error.empty()) break;
               break;
```

```
            }
            case AstNodeType_ACTIONS: {
                  //std::cout << "AstNodeType_ACTIONS" << std::endl;
                  execute_block(&program, child);
                  if (!program.error.empty()) break;
                  break;
            }

            default: {
                  program.error = "Unrecognized section";
                  if (!program.error.empty()) break;
                  break;
            }
            }

      }

      if (!program.error.empty()) {
            std::cout << "Runtime error: " << program.error << std::endl;
            return 1;
      }
      return 0;


}
```

Додаток 3
Копія презентації

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КОМП’ЮТЕРНИХ СИСТЕМ

КОМПІЛЯТОР МОВИ ASAMPL

Виконав: Песчанський Владислав Юрійович

Керівник: к.т.н. доцент Сулема Євгенія Станіславівна

Київ – 2019

ПОСТАНОВКА ЗАДАЧІ

Мета проекту: розроблення компілятора згідно формального опису граматики мови оброблення мультимедійних даних ASAMPL.

ЗАВДАННЯ

1. Проаналізувати доступні засоби для реалізації програмного забезпечення.

2. Розробити архітектуру програмного забезпечення.

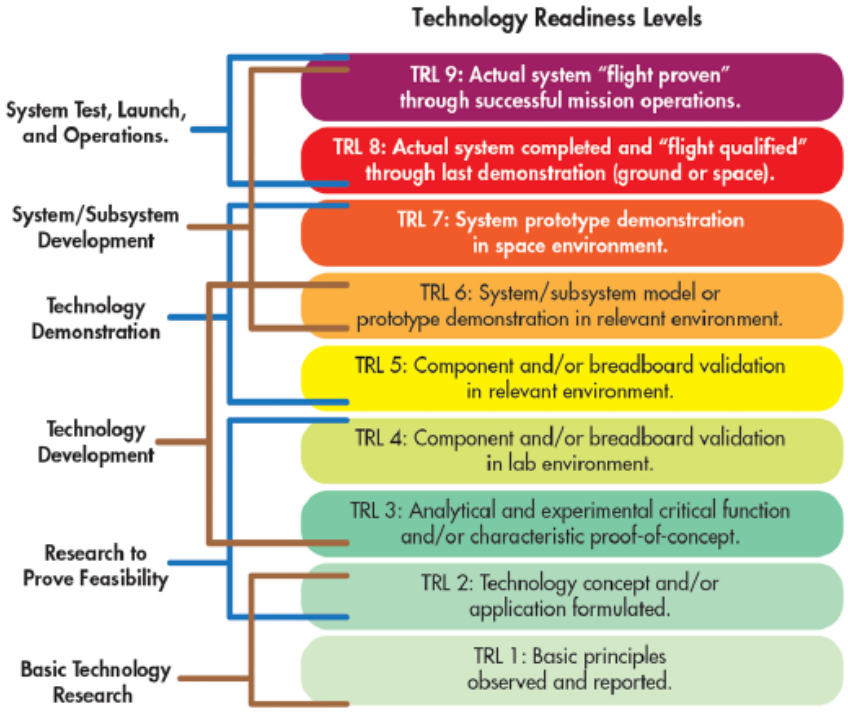
3. Розробити програмний продукт згідно з технічним завданням.

4. Протестувати розроблений програмний продукт.

АКТУАЛЬНІСТЬ

- На сьогодні не існує повнофункціонального компілятора мови програмування ASAMPL.

- Існуючі фреймфорки не призначені для роботи з широким спектром мультимедійних даних.



ОБГРУНТУВАННЯ ЗАСОБІВ РОЗРОБЛЕННЯ

- Було проведено порівняння таких обраних мов програмування: Java, C, C++, C#, Python.

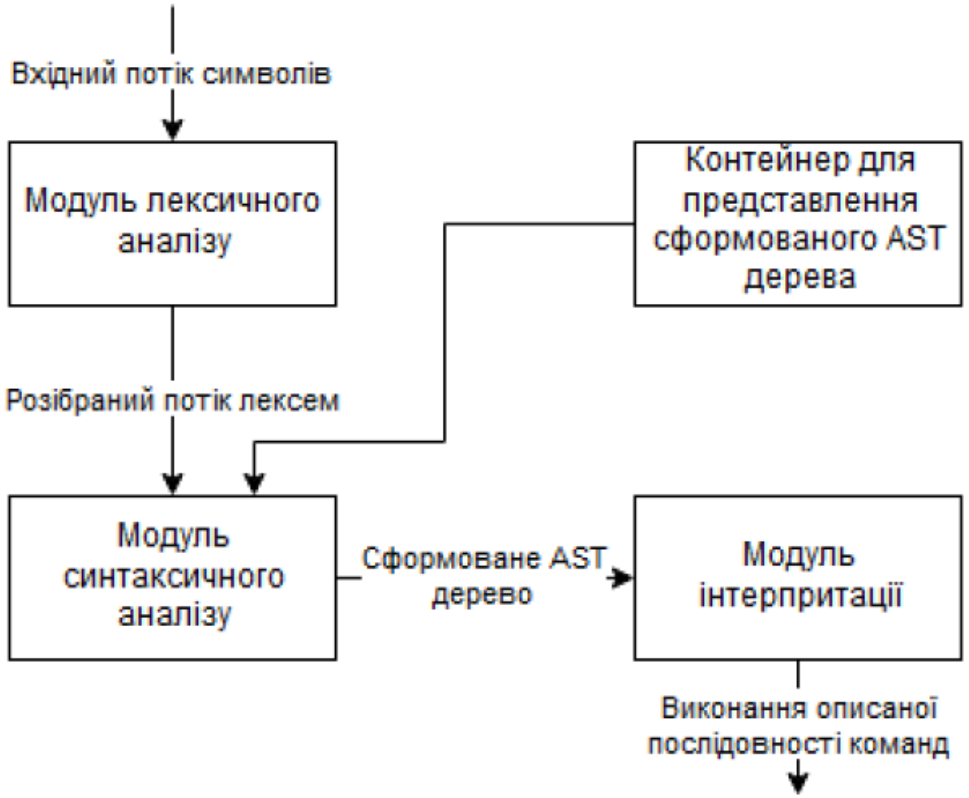
- Основні критерій порівняння: зручність синтаксису, гнучкість засобів для роботи з компіляторами та доступність навчальних засобів та документації.

- Висновок: за основний засіб розроблення слід обрати C++.

ОСНОВНІ МОДУЛІ ПРОГРАМНОГО ПРОДУКТУ

1. Модуль лексичного аналізу.
2. Модуль синтаксичного аналізу.
3. Модуль транслятору.

УЗАГАЛЬНЕНА СХЕМА ВЗАЄМОДІЇ МОДУЛІВ



МОДУЛЬ ЛЕКСИЧНОГО АНАЛІЗУ

- Для модуля лексичного аналізу вхідного потоку символів було розроблено алгоритм на основі машини станів.

- Розпізнані лексеми додаються до загального списку.

МОДУЛЬ СИНТАКСИЧНОГО АНАЛІЗУ

- Основним принципом роботи модулю синтаксичного аналізатору є розбір правил, реалізований за методом рекурсивного спуску.

- Застосовані правила послідовно, зліва направо поглинають лексеми, отримані від лексичного аналізатора.

МОДУЛЬ ТРАНСЛЯТОРУ

- Основним принципом роботи модулю трансляції є виконання пооператорної обробки вхідної послідовності команд, які представлені абстрактним синтаксичним деревом.

- Оброблені команди виконуються одразу після обробки їх модулем.

ПРИКЛАД ОБРОБЛЕННЯ ПОМИЛКИ ШЛЯХУ ДО ФАЙЛУ

test.txt — Блокнот

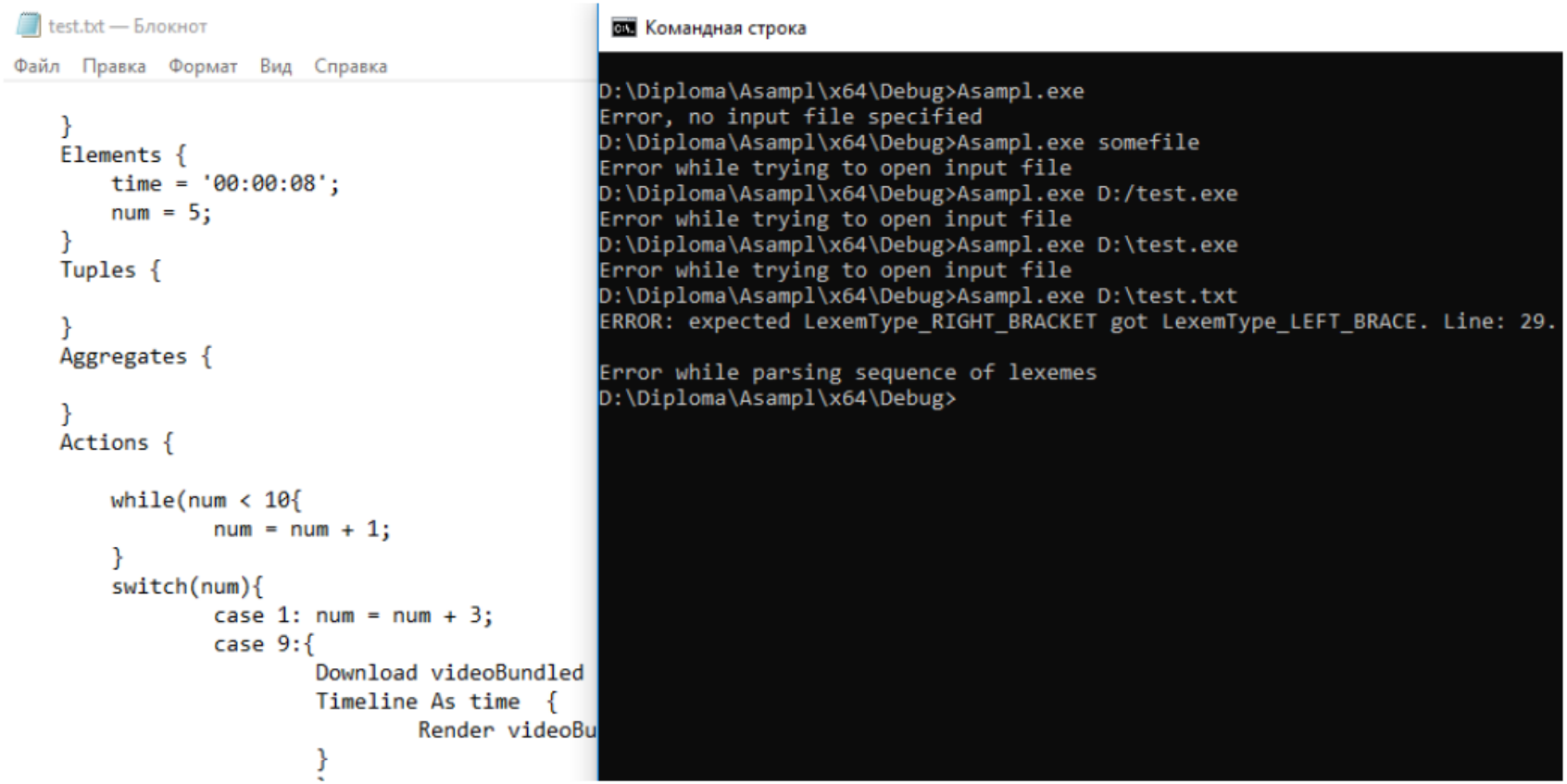
Файл Правка Формат Вид Справка

```
PROGRAM TestProgram{
    Libraries {
    }
    Handlers {

    }
    Renderers {

    }
    Sources {
        videoIn from 'E:/video.mp4';
        audioIn from '/tmp/data/in.ogg';
        audioOut from '/tmp/data/out.ogv';
    }
    Sets {

    }
    Elements {
        time = '00:00:08';
        num = 5;
    }
    Tuples {

    }
```

Командная строка

```
D:\Diploma\Asampl\x64\Debug>Asampl.exe
Error, no input file specified
D:\Diploma\Asampl\x64\Debug>Asampl.exe somefile
Error while trying to open input file
D:\Diploma\Asampl\x64\Debug>
```

ПРИКЛАД ОБРОБЛЕННЯ ПОМИЛКИ ПОБУДОВИ AST



ПРИКЛАД ОБРОБЛЕННЯ ПОМИЛКИ ПІД ЧАС ВИКОНАННЯ ПРОГРАМИ

test.txt — Блокнот

Файл Правка Формат Вид Справка

```
Sources {
    videoIn from 'E:/video.mp4';
    audioIn from '/tmp/data/in.ogg';
    audioOut from '/tmp/data/out.ogv';
}
Sets {

}
Elements {
    time = '00:00:08';
    num = 5;
    num2 = 7;
}
Tuples {

}
Aggregates {

}
Actions {

    num2 = num/0;
    print(num);
```

Командная строка

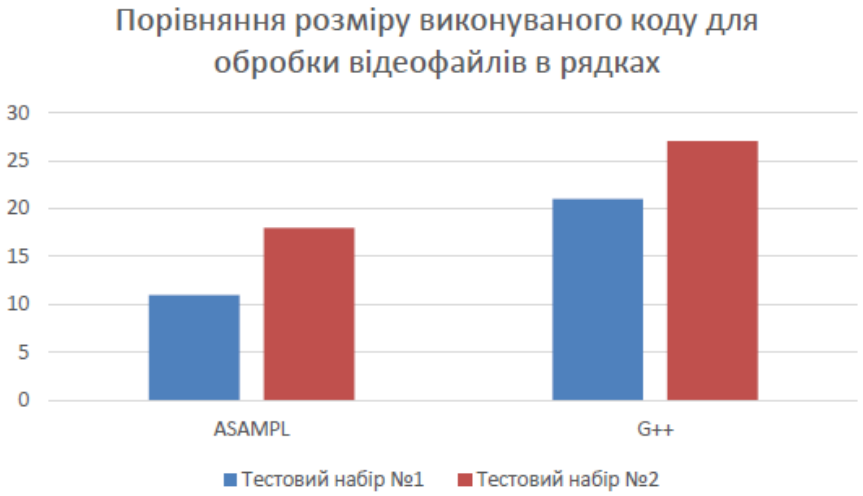
```
D:\Diploma\Asampl\x64\Debug>Asampl.exe
Error, no input file specified
D:\Diploma\Asampl\x64\Debug>Asampl.exe somefile
Error while trying to open input file
D:\Diploma\Asampl\x64\Debug>Asampl.exe D:/test.exe
Error while trying to open input file
D:\Diploma\Asampl\x64\Debug>Asampl.exe D:\test.exe
Error while trying to open input file
D:\Diploma\Asampl\x64\Debug>Asampl.exe D:\test.txt
ERROR: expected LexemType_RIGHT_BRACKET got LexemType

Error while parsing sequence of lexemes
D:\Diploma\Asampl\x64\Debug>Asampl.exe D:\test.txt
Runtime error: Invalid operation

D:\Diploma\Asampl\x64\Debug>_
```

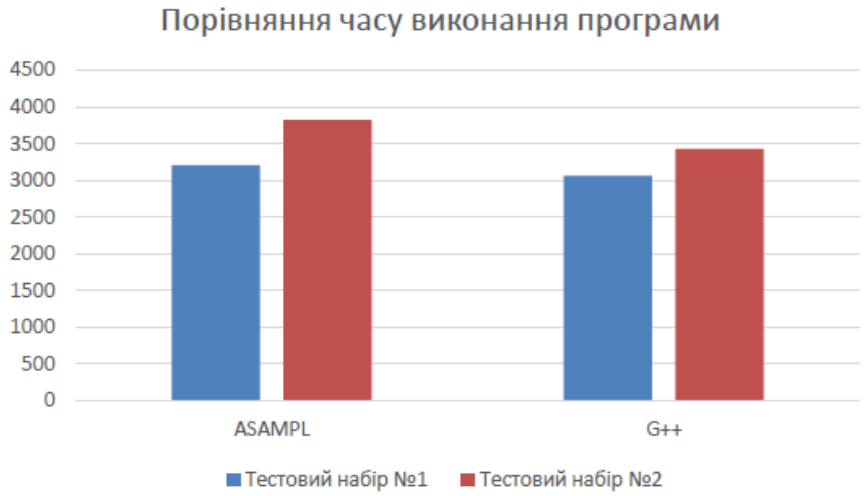
ПРИКЛАД ВИКОНАННЯ ПРОГРАМИ

КРИТЕРІЇ ОЦІНЮВАННЯ ЯКОСТІ ПРОГРАМНОГО ПРОДУКТУ

- Виконання програм, написаних згідно з формальним описом мови ASAMPL.
- Стійкість до помилок виникаючих під час виконання коду написаного мовою ASAMPL.
- Основні критерії порівняльного тестування: час виконання та кількість рядків коду для оброблення однієї послідовності дій мовою ASAMPL.

ГРАФІКИ РЕЗУЛЬТАТІВ ПОРІВНЯЛЬНОГО ТЕСТУВАННЯ





ВИСНОВКИ

1. Розроблено архітектуру ПЗ та виділено три основні модулі: лексер, парсер і транслятор.
2. Розроблено алгоритм лексичного аналізу на основі машини станів.
3. Адаптовано алгоритм рекурсивного спуску для побудови AST дерева для поданої граматики.
4. Розроблено алгоритм послідовного виконання команд занесених до AST для модуля транслятору.
5. Розроблено та протестовано компілятор мови обробки мультимедійних даних ASAMPL.

Дякую за увагу!

Додаток 4
Формальний опис граматики мови ASAMPL

Синтаксис мови ASAMPL (версія 1.5)

часовий_оператор = TIMELINE , (*ідентифікатор* | *значення_часу*) , ":", (*ідентифікатор* | *значення_часу*) , ":" , (*ідентифікатор* | *значення_часу*) , "{" , { *оператор* } , "}" | TIMELINE , AS , *кортеж_значень_часу* , "{" , { *оператор* } , "}" | TIMELINE , UNTIL , *логічний_вираз* , "{" , { *оператор* } , "}" ;
оператор_послідовної_обробки = SEQUENCE , "{" , { *оператор* , [";"]} , "}" ;
оператор_розгалуження = IF , *логічний_вираз* , THEN , "{" , { *оператор* } , "}" | IF , *логічний_вираз* , THEN , "{" , { *оператор* } , "}" , ELSE , "{" , { *оператор* } , "}" ;
оператор_вибору = CASE , *ідентифікатор* , OF , "{" , { (*ідентифікатор* | *значення*) , ":" , *оператор* } , "}" | CASE , *ідентифікатор* , OF , "{" , { (*ідентифікатор* | *значення*) , ":" , *оператор* } , "}" , ELSE , "{" , (*ідентифікатор* | *значення*) , ":" , *оператор* , "}" ;
оператор_заміни = SUBSTITUTE , *ідентифікатор* , FOR , *ідентифікатор* , WHEN *логічний_вираз* ;
оператор_завантаження = DOWNLOAD , *ідентифікатор* , FROM , *ідентифікатор* [WITH , *ідентифікатор*] | UPLOAD , *ідентифікатор* , TO , *ідентифікатор* [WITH , *ідентифікатор*] ;
оператор_відтворення = RENDER , *ідентифікатор* , WITH , *ідентифікатор* ;
оператор_присвоювання = *ідентифікатор*, (IS | "=") , *значення* ;
оператор = *часовий_оператор* | *оператор_послідовної_обробки* | *оператор_розгалуження* | *оператор_вибору* | *оператор_заміни* | *оператор_завантаження*

| *оператор_присвоювання* | *оператор_відтворення* | *операнд* , *операція* , *операнд* ;

операнд = *ідентифікатор* | *значення* ;

операція = *логічна_операція* | *логічна_операція_над_агрегатом* | *математична_операція*;

математична_операція = "+" | "-" | "*" | "/" | "^" ;

логічна_операція = "~" | "&" | "|" ;

логічна_операція_над_агрегатом = "∪" | "∩" | "\" | "Δ" ;

тип = *тип_абсолютний_час* | *тип_відносний_час* | *тип_абсолютна_дата* | *тип_відносна_дата* | *чисельний_тип* | тип_*кадр* | *текстовий_тип* | *посилаьний_тип* | *логічний_тип* | *тип_подія* ;

тип_множини = *тип* ;

тип_абсолютний_час = ATIME ;

тип_відносний_час = RTIME ;

тип_абсолютна_дата = ADATE ;

тип_відносна_дата = RDATE ;

чисельний_тип = INTEGER | REAL | DOUBLE ;

тип_кадр = FRAME ;

текстовий_тип = TEXT ;

посилаьний_тип = LINK ;

логічний_тип = LOGIC ;

тип_подія = EVENT ;

значення = *значення_абсолютного_часу* | *значення_відносного_часу* | *значення_абсолютної_дати* | *значення_відносної_дати* | *число* | *кадр* | *текстова_послідовність* | *посилання* | *логічне_значення* | *подія* ;

значення_абсолютного_часу = [*цифра* , { *цифра* } , ":" ,] *цифра* , [*цифра*] , ":" , *цифра* , [*цифра*] , [":" , *цифра* , [*цифра*] , "." , *цифра*] , [("+" | "-") , *цифра* , *цифра* , ":" , *цифра* , *цифра*] ;
значення_відносного_часу = [("+" | "-")] , [*цифра* , { *цифра* } , ":"] *цифра* , { *цифра* } , ":" , *цифра* , { *цифра* } [, ":" , *цифра* , { *цифра* }] ;
значення_абсолютної_дати = [*цифра* , *цифра* , [*цифра* , *цифра*] , "/" ,] *цифра* , [*цифра*] , "/" , *цифра* [, *цифра*] ;
значення_відносної_дати = [("+" | "-")] , [*цифра* , *цифра* , { *цифра* , *цифра* } , "/" ,] *цифра* , { *цифра* } , "/" , *цифра* , { *цифра* } ;
число = *ціле_число* | *дійсне_число* | *дійсне_число_подвійної_точності* ;
ціле_число = ["-"] , *цифра* , { *цифра* } ;
дійсне_число = ["-"] , *цифра* , { *цифра* } , "." , *цифра* , { *цифра* } ;
дійсне_число_подвійної_точності = ["-"] , *цифра* , "." , *цифра* , { *цифра* } , "e" , ("-" | "+") , *цифра* , { *цифра* } ;
кадр = "[" , *число* , { "," , *число* } , { ";" , *число* , { "," , *число* } } "]" ;
текстова_послідовність = "”" , { *літера* | *символ* | *цифра* }, "”" ;
посилання = "’" , *літера* , *літера* , *літера* , [*літера* | *символ* | *цифра*] "://" , (*літера* | *символ* | *цифра*) { *літера* | *символ* | *цифра* } , "." , (*літера* | *символ* | *цифра*) { *літера* | *символ* | *цифра* } , { "/" , (*літера* | *символ* | *цифра*) { *літера* | *символ* | *цифра* } , "." , (*літера* | *символ* | *цифра*) { *літера* | *символ* | *цифра* } } , "’" | "’" , *літера* , ":\" , (*літера* | *символ* | *цифра*) { *літера* | *символ* | *цифра* } , { "\" , (*літера* | *символ* | *цифра*) { *літера* | *символ* | *цифра* } } , "’" ;
логічне_значення = TRUE | FALSE ;
подія = ідентифікатор ;

літера = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" ;
цифра = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
символ = "." | "," | ";" | ":" | "!" | "?" | "(" | ")" | "[" | "]" | "{" | "}" | "<" | ">" | "’" | "”" | "+" | "-" | "*" | "/" | "=" | "\" | "@" | "#" | "$" | "|" | "%" | "^" | "&" ;
ідентифікатор = літера , { *літера* | *цифра* } ;
кортеж_значень = ідентифікатор | "[" , { *значення* [","] } , "]" ;
кортеж_значень_часу = ідентифікатор | "[" , { *значення_часу* [","] } , "]" ;
логічний_вираз = ідентифікатор | *ідентифікатор* , *логічна_операція* , *ідентифікатор* | *логічне_значення* ;
коментар = текстова_послідовність ;
програма = PROGRAM , *ідентифікатор* , "{" , LIBRARIES , "{" , { *ідентифікатор* , (IS | "=") , *посилання* , ";" } ["//" , *коментар*] "}" , , HANDLERS , "{" , { *ідентифікатор* , (IS | "=") , *посилання* , ";" } ["//" , *коментар*] "}" , , RENDERERS , "{" , { *ідентифікатор* , (IS | "=") , *посилання* , ";" } ["//" , *коментар*] "}" , SOURCES , "{" , { *ідентифікатор* , (IS | "=") , *посилання* , ";" } ["//" , *коментар*] "}" , SETS , "{" , { *ідентифікатор* , (IS | "=") , *тип* , ";" } ["//" , *коментар*] "}" , ELEMENTS , "{" , { *ідентифікатор* , (IS | "=") , (

тип_множини | *значення*) , ";" } ["//" , *коментар*] "}" ,
TUPLES , "{" , { *ідентифікатор* , (IS | "=") , *тип_множини* ,
";" } ["//" , *коментар*] "}" ,
AGGREGATES "{" , { *ідентифікатор* , (IS | "=") , "[" ,
(*ідентифікатор* | *кортеж_значень* , ";" } ["//" , *коментар*
] "}" ,
ACTIONS , "{" , { *оператор* , ";" } ["//" , *коментар*] "}" , ["//" ,
коментар] , "}" ;

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

"ЗАТВЕРДЖЕНО"

Науковий керівник кафедри

_____ І.А. Дичка

"___" _____ 2018 р.

КОМПІЛЯТОР МОВИ ASAMPL

Програма та методика тестування

ДП.045480-04-51

"ПОГОДЖЕНО"

Керівник проекту:

_____ Є.С. Сулема

Нормоконтроль:

_____ М.В. Онай

Виконавець:

_____ Песчанський В.Ю.

2018

ЗМІСТ

1. ОБ’ЄКТ ВИПРОБУВАНЬ

Компілятор мови обробки мультимедійних даних ASAMPL, написаний мовою C++.

2. МЕТА ТЕСТУВАННЯ

У процесі тестування має бути перевірено наступне:

1. Функціональна працездатність компонентів додатку;
2. Відповідність розпізнаваної мови розробленій граматиці з врахуванням семантики мовних конструкцій;
3. Відповідність вимогам Технічного завдання.

3. МЕТОДИ ТЕСТУВАННЯ

Тестування виконується методом Gray Box Testing. На відповідність функціональним вимогам перевіряються як окремі компоненти, так і безпосередньо програмний продукт загалом. Використовуються наступні методи:

1. Функціональне тестування, зокрема на рівні Critical path – тестування основних функцій;
2. Димове тестування;
3. Тестування інтерфейсу.

4. ЗАСОБИ ТА ПОРЯДОК ТЕСТУВАННЯ

Працездатність розробленого додатку перевіряється з використанням:

1. Покриття юніт-тестами основних компонентів з використанням бібліотеки unittest;
2. Статичного аналізу коду;
3. Для системи в цілому – набору файлів з вихідним кодом мовою ASAMPL, що подаються на вхід, та очікуваного виведення програми до них;
4. Ручного тестування.

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

"ЗАТВЕРДЖЕНО"

Науковий керівник кафедри

_____ І.А. Дичка

"___" _____ 2019 р.

КОМПІЛЯТОР МОВИ ASAMPL

Керівництво користувача

ДП.045480-05-34

"ПОГОДЖЕНО"

Керівник проекту:

_____ Є.С. Сулема

Нормоконтроль:

_____ М.В. Онай

Виконавець:

_____ Песчанський В.Ю.

2019

ЗМІСТ

1. НАЛАШТУВАННЯ СЕРЕДОВИЩА ВИКОНАННЯ

Для встановлення компілятору необхідно:

1. Завантажити графічну бібліотеку OpenCV, для мови C++.
2. Розмістити завантажену бібліотеку у файловій системі комп’ютера.
3. Додати шлях до завантаженої бібліотеки у список змінних середовищ так, як зображено на рис. 1.
4. Додати шлях до завантаженої бібліотеки до додаткових директорій середовища Visual Studio так, як зображено на рис. 2.
5. Додати шлях до завантаженої бібліотеки до лінкеру середовища Visual Studio так, як зображено на рис. 3.
6. Додати шлях до завантаженої бібліотеки до вводу лінкеру середовища Visual Studio так, як зображено на рис. 4.

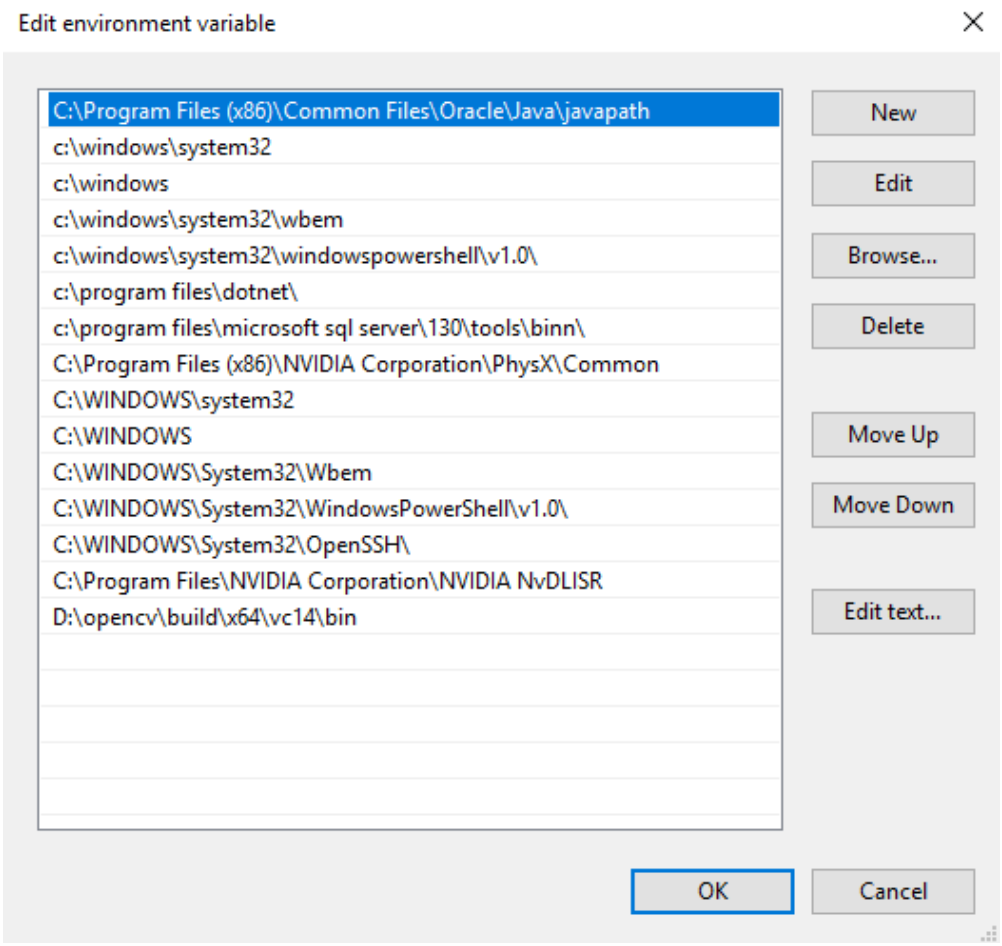


Рис. 1. Список змінних середовищ

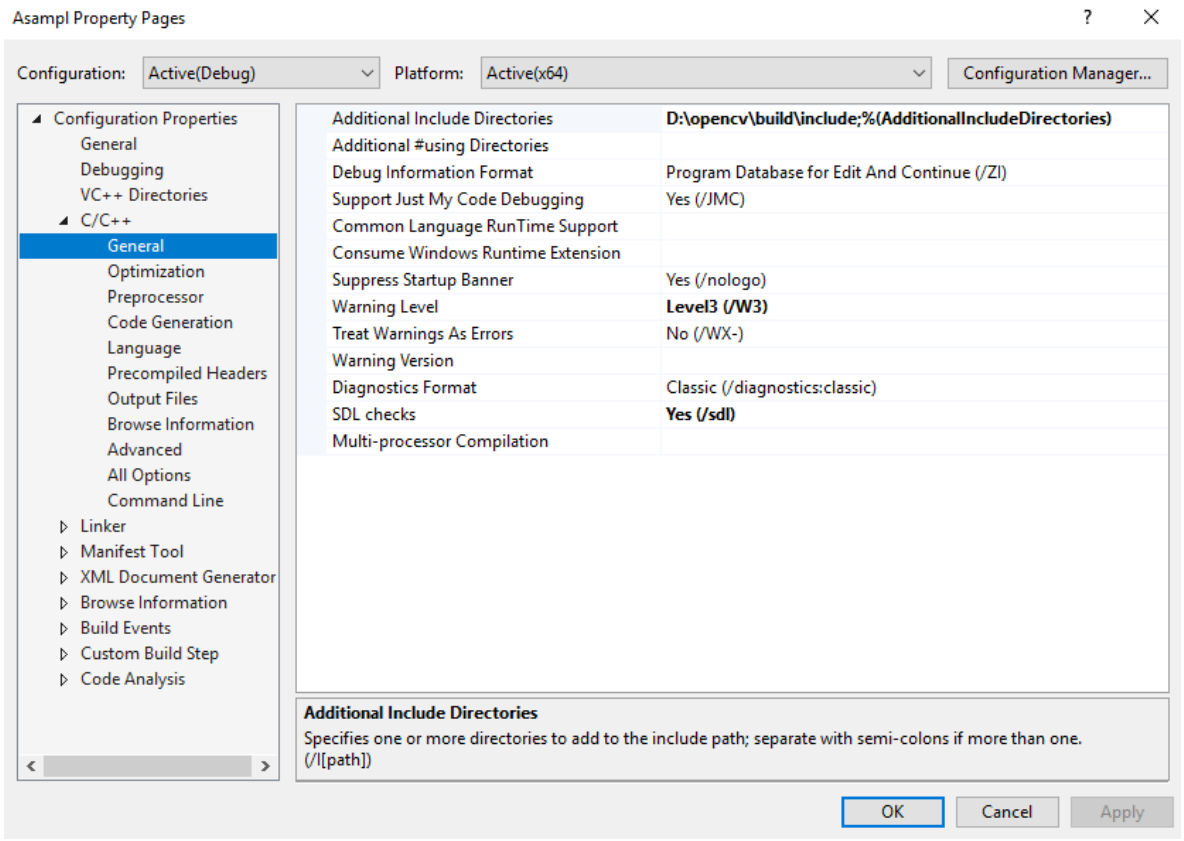


Рис. 2. Список додаткових директорій

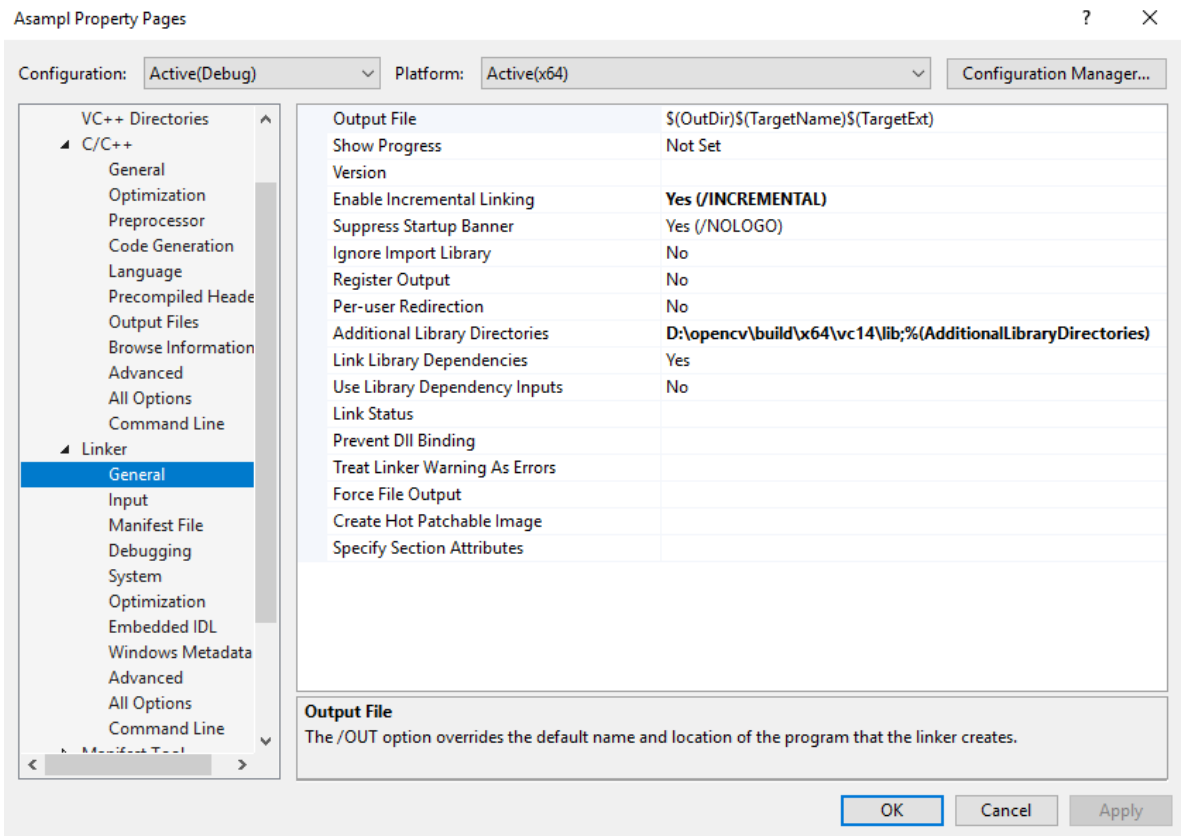


Рис. 3. Список додаткових бібліотек

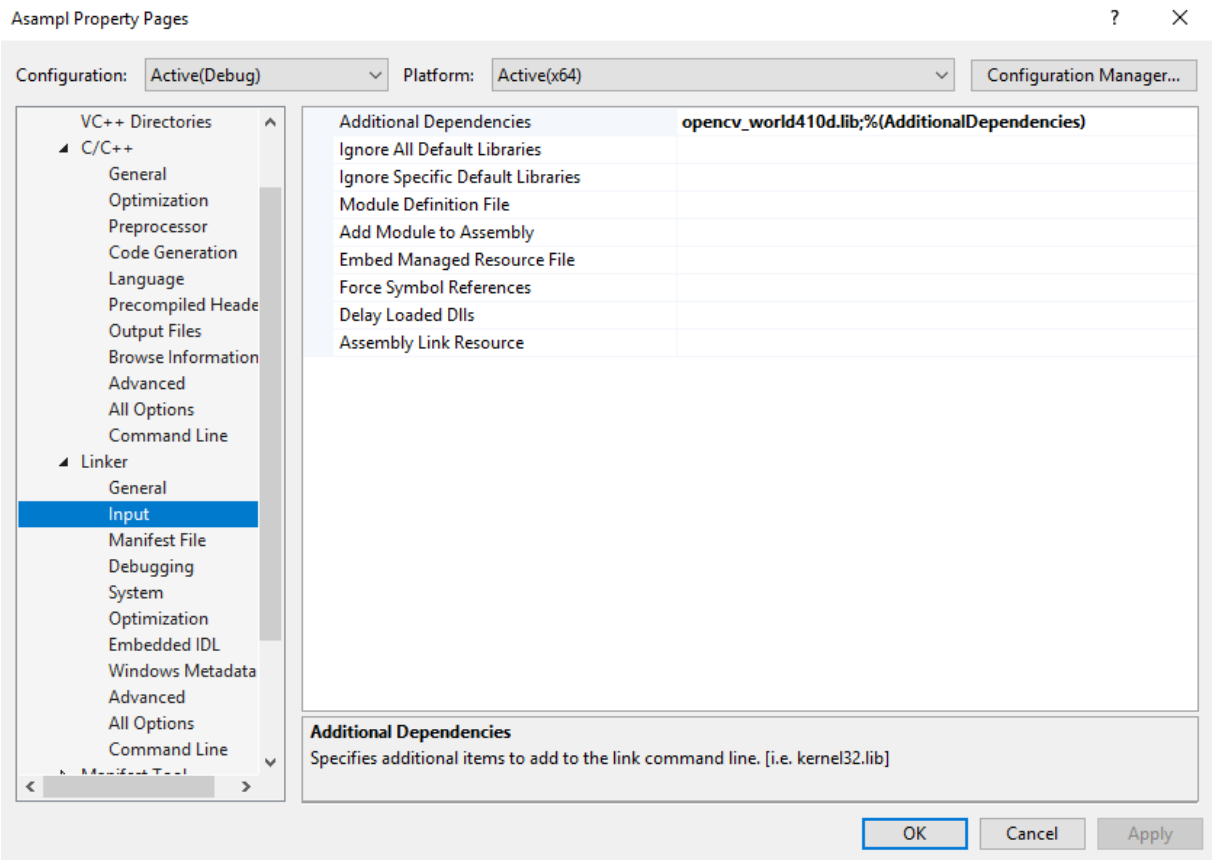


Рис. 4. Додаткові бібліотеки вводу

2. ОПИС ПОСЛІДОВНОСТІ ДІЙ ДЛЯ ВИКОНАННЯ ПРОГРАМИ

Для виконання послідовності команд, написаних мовою ASAMPL необхідно:

1. Створити текстовий файл, зміст якого буде написаний з урахуванням формального опису граматики мови ASAMPL.
2. Запустити файл компілятору мови ASAMPL вказавши шлях до файлу у аргументах командного рядка.